A supportive environment for the management of software testing

Liu, Lulu

A Support Environment for the Management of Software Testing

Lulu Liu

Dissertation submitted for the degree of
Master of Science

Computer Science
School of Engineering and Computer Science
University of Durham

May 1992

Dedicated to my grandparents who brought me up

Abstract

This dissertation describes research undertaken on the management of software testing. A support environment for the management of software testing, entitled SEMST, is presented. The research approach involves the investigation of software configuration management and its application to the testing process; the study of software testing techniques and methods; the exploration of the significance of software testing management; a survey of related work; the development and analysis of the requirements for SEMST; its implementation and an assessment. The current version of SEMST is a prototype built on the top of Unix and RCS on a Sun workstation. It is able to maintain all versions of specifications, test cases and programs, as well as to manage the relationships between these components.

Acknowledgements

This is a revision to the version submitted in January 1991. During the modification, I obtained financial support and resources from The Polytechnic of Central London, School of Computer Science & Information Systems Engineering. I am extremely grateful to them. The original work was funded by a European Commission Grant under the Esprit II Programme – REDO. I would like to thank the University of Durham, Department of Computer Science for this grant and the facilities provided. I am deeply indebted to my supervisor Dr. D. J. Robson for his advice and patience while supervising me on this work. I would also like to express my great gratitude to Prof. K. H. Bennett and Dr. G. Edmunds for their careful reading of this work and constructive criticisms that significantly helped to improve the previous version of the text. In the course of writing this dissertation, I have received assistance from a number of people. In particular, thanks are due to Rachel Kenning who helped me in understanding software configuration management. Finally, I thank my sister and my parents for their concern and encouragement when I was doing this work. My good will goes to all of them.

Contents

Chapter 1

Introduction

The research described in this dissertation is intented to apply software configuration management methods and database support techniques to the testing process. In the area of software engineering, particularly in the context of software testing, this research can be classified as an activity which is investigating the support techniques and methods for test data control and management.

This chapter is an introduction to the research. Section 1.1 presents a conceptual overview of software engineering, software configuration management and software testing. Section 1.2 describes the evolution of software testing, and section 1.3 addresses the objectives of this research. The structure of the dissertation is presented in the last section.

1.1 Overview

The term *software engineering* was first introduced in the late 1960s at a NATO conference [82] held to discuss what had been described as the *software crisis.* The software crisis is a term used to cover the problems of the production of reliable and maintainable software on schedule. The problems involved in the construction of large software systems are immense so that software engineering is concerned with the activity of developing and maintaining large software system. According to [33], "software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates". The primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of software engineers. In the construction of a large software system, a number of distinct stages can be identified. These make up what is known as the *software life cycle.*

Basic to the concept of the software engineering is the need to manage and control all software components developed, used and modified during the software life cycle and to ensure a correct software product is produced. *Software configuration management* is a method which applies an engineering approach to tracking and controlling the evolution of software components. *Software testing* is a method used to assess and improve the quality of software. It is viewed as the continuous task of planning, designing, and constructing tests, and of using these tests to assess and evaluate the quality of work performed at each step of the system development. The term 'software testing' has been used broadly to include the full scope of what is sometimes referred to as test and evaluation or verification and validation activities.

It has been said that [81] "approximately 50% of the elapsed time and over 50% of the total cost are expended in testing a program or a system being developed". As a large proportion of the total software expense is spent on software testing, this area has considerable potential for reducing the cost of software production.

1.2 Software Testing: A Historical Perspective

The notion of testing programs arose almost simultaneously with the construction of the first program. According to the early view of software testing a program is first written then tested and debugged. This view considers testing a *follow on* activity and embraces the effort not only to discover errors but also to correct and remove them. A number of the earliest papers on testing actually address "debugging". It was not until 1957 that program testing was clearly distinguished from debugging [50].

During the late 1950s and 1960s, software testing came to assume more and more significance because of both experience and economics. It was evident that computer systems contained many deficiencies, and the cost of the recovering from these problems were substantial. Because of that, more emphasis was placed on "better testing" by users and project managers.

The first formal conference on software testing was held in June 1972 at the University Of North Carolina. *Program Test Methods* written by William Hetzel was published as a result of this conference and established the view that "testing encompassed a wide array of activities all associated with obtaining confidence that a program or system performed as it was supposed to" [49].

Since that initial conference many conferences and workshops have been devoted to software quality, reliability, and engineering. Gradually the "testing discipline" has emerged as an organised element within software technology, and testing technology has been given individual emphasis in software development.

During the last few years a number of books on testing have contributed to this growing technology [81] [4]. Many programming and project management texts have included several chapters on testing, and testing basics are taught in most programming courses.

However, the testing field is far from mature. Even satisfactory agreement on a def-

inition of testing still remains in question. The following is the traditional definition of testing, which was made by Myers [81], and is still supported by some people: "Testing is the process of executing a program with the intent of finding errors."

This view of testing makes "finding errors" the goal. Myers thinks that people should start with the assumption that the program contains errors and then test it to find as many of the errors as possible. He states that "if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors and we become more successful in testing [81]."

While Myers' definition and its implications have been important in understanding testing, it has frequently been argued by many researchers that it is too narrow and restrictive to accept as a definition of testing. The disagreement centres on finding errors as the goal. Weyuker and Ostrand in [118] introduced the notion of error-based testing which is performed with the aim of eliminating errors in the programs. They believe that a careful study of a test method can uncover the class of errors detectable by the method. If the method fails to detect any errors, one can conclude that all errors in the detectable class are absent.

Hetzel [50] gave another view of testing, which I shall use as the definition of testing in this dissertation:

> Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

Two terms often associated with testing are *verification* and *validation.* Verification refers to ensuring correctness from phase to phase of the software life cycle. Validation involves checking the the software against the requirements. Sometimes, verification is associated with formal proofs of correctness, while validation is concerned with executing the software with test data. Under the present state of software quality control technology,

testing is widely believed to be an important pragmatic verification and validation mechanism for a software project. Therefore, as Hetzel says [50], "testing should be looked at as a broad and continuous activity throughout the development process. Any activity that is undertaken with the objective of evaluating or measuring an attribute of the software should be considered a testing activity. This includes most reviews, walk-throughs, and inspections, and many judgements and analyses that are performed."

For the past two decades, there have been a wide range of research and development activities in software testing. These activities can be classified as follows:

1. **Establishment of testing theory.** This research focuses on the criteria for adequate and reliable testing.
2. **Exploration of new testing techniques.** This activity is concerned with the study and development of the requirements for new testing tools and techniques. It is the area in which most researchers have put their efforts in the past.
3. **Assessment of existing testing techniques.** This involves the evaluation and comparison of the effectiveness of various testing methods.
4. **Management of productive testing processes.** This is associated with the research into controlling and managing the testing organisation, resources, scheduling and ensuring a smooth data flow through the whole process of testing.

The research described in this dissertation belongs to the fourth category.

1.3 Objectives of the Research

There is no doubt that even use of the best design and requirements methods and of the best testing techniques will not result in the construction of a cost-effective and high qual-

ity system if the testing process is poorly managed. Unfortunately, in the past years little attention has been given to the management of the testing process. Although there has been significant progress in the development of database support techniques and project management tools which provide systematic approaches to managing a software development process, the management mechanisms for the testing process is not well-furnished in these tools. The growth of testing techniques and methods requires more management and control in testing.

The research effort described in this dissertation is devoted to developing a database support system for the management of software testing. The intention is to apply software configuration management method to the software testing process. The following is a list of the research objectives:

- The study of software configuration management methods.
- The study of software testing principles and techniques.
- An investigation into the management of software testing.
- A survey and evaluation of previous work in software testing management.
- The establishment and analysis of the requirements for the new system to be developed.
- The design and implementation of the system prototype.
- An assessment of the developed new system.
- A discussion of possible future extensions.

1.4 Structure of the Dissertation

This dissertation consists of seven chapters.

Chapter 2 describes the concepts of software configuration management as well as existing tools and techniques in this area; the project requirements are proposed in this chapter. Chapter 3 presents an overview and analysis of software testing techniques; it focuses on a discussion of specification-based/program-based testing techniques and regression testing techniques. Chapter 4 is concerned with software testing management, in which the testing activities in the software life cycle, the need for testing management, and configuration management in software testing are described. Chapter 5 presents an investigation of previous work and an analysis of the requirements for the new system. Chapter 6 introduces the new system developed for supporting testing management. Chapter 7 presents an assessment of the system prototype; discusses possible future research work in this area; and summarises the results of the research.

The use of the new system is described in Appendix A.

Chapter 2

Software Configuration Management

Introduction

This chapter describes *Software Configuration Management* (SCM) – the discipline of controlling the evolution of software systems. SCM applies to all representations of the software system from requirements through to executable code and is based on four sub-disciplines: software configuration identification; software configuration control; software configuration status accounting; and software configuration audit.

The chapter is organised into three main parts. The first part, consisting of sections 2.1 and 2.2, addresses the concepts of SCM and the reasons for its existence. The second part, comprising sections 2.3 and 2.4, describes several existing SCM tools from a technical perspective. The third part is section 2.5 which summarises the descriptions in this chapter and presents the project requirements.

2.1 What is SCM?

2.1.1 Origins of CM

The term *configuration management* derives from hard engineering disciplines, such as mechanical, electrical and industrial engineering, which use change control techniques to manage blueprints and other design documents [2]. These techniques were then used to bring computer hardware production under configuration management (CM) control. As the complexity and lengthy time scales of software production increased, the use of configuration management was expanded to include software.

One of the earliest definitions of configuration management was given by the Department of Defense (DoD) in the 1968 Military Standard (MIL-STD) 480 [24]. This standard was later revised to include the management of software and was re-issued in 1970 as MIL-STD-483 to describe the use of configuration management techniques in the production of systems for the military [25] [3].

In 1983 an IEEE standard for configuration management was produced [54], which outlines the structure for developing a configuration management plan.

From the SCM point of view, the software product is not a single system, but a set of similar configurations, organised in families. SCM differs from CM in that software is easier to change and therefore changes more rapidly than hardware; and in that SCM is potentially more easily automated because all components of a software system are stored online.

Initially, SCM was a manual set of procedures, mainly used as a management discipline. It has been more recently developed as an automated procedure used for both technical and management practice, and is the subject of increasing interest because of its use in integrated programming support environments.

2.1.2 Definition of SCM

According to [54], the definition of SCM is:

> The process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items.

There are also a number of similar definitions of SCM. A collection of these definitions can be found in [64].

2.1.3 The Problems

The importance of SCM for managing and maintaining large software projects, and for coordinating software development to minimise confusion caused by interaction among team members is now recognised. Babich [2] illustrates this by describing three problems associated with software development.

1. **The Double Maintenance Problem.** Double maintenance is the problem of keeping multiple identical copies of software. When there are two copies of the same software, then both copies need to be maintained. When a bug is found, it must be fixed identically in both copies.

2. **The Shared Data Problem.** This problem arises when the software consists of a single entity and many people are simultaneously accessing and modifying it. Thus the changes made by one programmer can interfere with the progress of others. The most obvious case of interference is when the modification is wrong.

3. **The Simultaneous Update Problem.** One solution to the above problems is to divide the software into a set of modules and store these modules in a shared data area. When a module needs to be modified, a copy of the module should be made and taken away from the shared data area. After the modification the validity of the changed module must be ensured before storing it in the shared data area as a replacement of the previous version of the module. However, a new problem may arise when two or more people update the copy of the same module at the same time. It is possible that someone's modification will get overwritten.

2.2 Software Configuration Concepts

This section presents the fundamental concepts of SCM. The concepts addressed in this section are based on the descriptions by Babich [2], Pressman[92] and Kenning[64].

2.2.1 Configuration

A *configuration* is the set of objects from which the software system is composed. Figure 1 shows the configuration of two different versions, named *version1* and *version2*. Objects A, B, D, and F appear in both configurations.

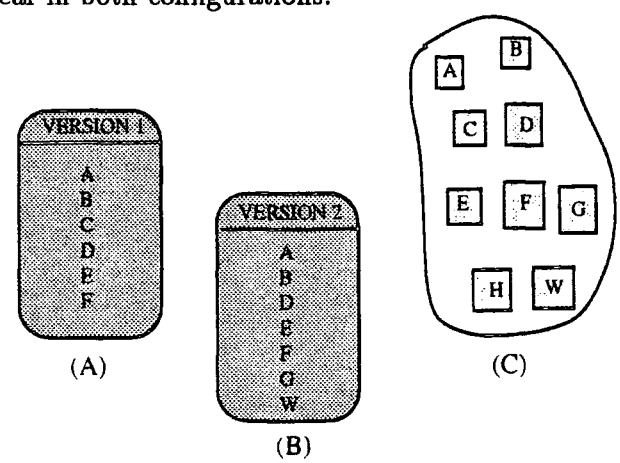

Figure 1. Configuration Concept

Choosing which items to go into a particular software product is called *configuring* the software. In Figure 1, (C) is called a library or a database.

Versions

Versions arise from changes made to software objects as a result of the need to correct, adapt or enhance the software. A source version group is the collection of interrelated source objects resulting from changes to a particular component in the system. Versions may be revisions or variants.

Revisions

Two versions of an object may differ because one is a revision of another. A revision is produced by changing an earlier version of a software object such that the new version is intended to supersede the old.

Variants

Unlike revisions, new variants do not supersede the old. Multiple variants coexist as equal alternatives by having the same functionality but for slightly different situations. Variants are named, not numbered, because there is no meaningful linear order among them. Thus the name of a variation reflects the purpose it serves, not the order in which it was created.

Derivations

The history of a software item is called its derivation. The purpose of the derivation is to record precisely and accurately all the information tracing the evolution of a software configuration item. Each software configuration item has a derivation and each derivation

references other software configuration items and therefore other derivations rather like a family tree. Derivations should identify the tool, the input to the tool, the options, the author and the reason for the change.

2.2.2 Baselines

A baseline in the context of SCM is defined as a milestone in the development of software that is marked by the delivery of a software configuration item, and the approval of this item by formal technical review. For instance, a preliminary design has been documented and reviewed; errors are found and corrected; the approved preliminary design becomes a baseline. Further changes to this item can now only be made after each change has been evaluated and approved.

2.2.3 Software Configuration Items

Software configuration items are the information created in a software development process. A software configuration item could be a single section of a large specification, a test case in a large suite of tests, a document, or a named program component (e.g. a Pascal procedure or an Ada package).

The following items, which can form a set of baselines [92] are considered as targets for configuration management techniques, and are examples of so-called *software objects*:

- System Specification
- Software Project Plan
- Requirements Documents
  (a) Software Requirements Specification

(b) Executable or Paper Prototype

- Preliminary User Manual
- Design Specification
  (a) Preliminary Design
  (b) Detail Design
- Source Code Listing
- Test Design Specification
  (a) Test Plan and Procedure
  (b) Test Cases and Recorded Results
- Operation and Installation Manual
- Executable Programs
- As-built User Manual
- Maintenance Documents
  (a) Software Problem Reports
  (b) Maintenance Requests
  (c) Engineering Change Orders
- Standards and Procedures for Software Engineering

2.2.4 SCM database

Tichy in [115] defines the basic elements of a database for SCM. This database stores all software objects produced during the project life cycle. Every object in the database has a unique identifier and a body containing the actual information. A set of attributes associated with the objects and a facility for linking objects via various relations are also needed. The set of attributes and relations must be extensible in a SCM database.

2.2.5 SCM Constitution

SCM is composed of four components:

1. **Configuration Identification.** This involves naming all items and baselines in the software configuration. The term "naming" refers to an identification scheme that provides the following information:

   - software configuration item type (e.g. document, program, test case);
   - software configuration item name;
   - project or product identification;
   - version number;
   - last release date;

   The identification data may be maintained in an automated database so that all relevant software configuration items for a specific version of software may be retrieved when requested.

2. **Configuration Control.** This is the systematic evaluation, coordination, and approval or rejection of proposed changes to the design and construction of the software, that have been requested by the development team, support group, or users.

3. **Configuration Status Accounting.** This is the recording and reporting of the identities and descriptions of all the software configuration items in the system, together with records of the status of proposed changes and the implementation state of approved changes. That is, it provides an administrative history of the way in which the system has evolved.

4. **Configuration Audits and Reviews.** These ensure compliance with configuration management requirements. Configuration management requires evidence that certain reviews and audits have been passed before products can be accepted.

2.3 Overview of Software Configuration Management Tools

In the past years, many tools have been developed to aid the SCM process. These tools are categorised [64] as single function tools such as *SCCS* [100] and *RCS* [112] for change and version control and *MAKE* [34] for program building; complete configuration management systems such as *CCC* [105] and *Lifespan* [90], and comprehensive programming environments such as *Gandalf* [61], *DSEE* [69] and *Adele* [30]. Most of these tools deal with SCM on the UNIX operating system and the literature is dominated by discussions centred around Unix.

MAKE is the 'original' program build tool in the family of SCM tools. It has been in use on Unix since 1975. MAKE provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It essentially performs the effects of a complete build without the cost of rebuilding files that are correct to begin with. The input to the MAKE tool is a file named '*Makefile*' which specifies for each object module, the module on which it depends and the Unix commands required to perform the functions on the objects. The Makefile is commonly stored under SCCS and RCS control. There are however, a number of limitations to MAKE [115]. For instance, MAKE only considers the most recent versions of a configuration, it does not maintain information of previous configuration, detection is based only on timestamps, and there is not easy integration with file archive and version control tools.

SCCS(The Source Code Control System) was developed by Rochkind in the early 70's [3] [100] and is distributed with most AT&T-derived versions of Unix. The purpose of SCCS is to control the baseline of source code for a software project. It can also be used to control baselines of documentation, tests, or other textual data. Each of the data files in the baseline may have multiple revisions and variants. SCCS manages all the files and can produce any versions of any file on demand. The 'charge-in' and 'charge-out' facilities are available so that team members may transfer objects into their working spaces (i.e. directories) for update. However, SCCS is weak in naming of versions when there are a

great number of variations of the objects, and it does not support merging of development paths [2].

RCS(The Revision Control System) is a similar system to SCCS, but it has some capabilities that SCCS lacks. It was developed by Tichy in 1982 [112] with the intention of improving on some of the inadequacies of SCCS. For instance, RCS provides support for semiautomatic merging of parallel development paths using a line-by-line comparison of the different versions, a facility not included in SCCS. The descriptions of RCS functions are presented in the next section. Both SCCS and RCS do not however provide management of object code, load image or other non-ASCII files.

Gandalf [61] is a software development environment which integrates the notions of programming and systems development. It consists of three components: System Version Control(SVCE), Incremental Program Construction and Project Management. These components operate on a common database through a uniform user interface provided by a syntax-directed editor. The SVCE provides a system generation facility based on system descriptions which include descriptions of system and subsystem interfaces, interdependencies, and parallel and successive versions. The SVCE keeps track of the location and status of all system objects, including the source programs, and can automatically generate an executable system from a system description.

Adele [30] is a programming environment developed at the University of Grenoble. It is independent of programming languages and operating systems, running on Unix, VAX/VMS and MS-DOS. Adele has four main components: a program editor, compiler and debugger; a parametrised code generator; a user interface; and a program base. The program base is used to support a configuration management system. It borrows some ideas of constructing tools and the program base from the Gandalf system.

DSEE(DOMAIN Software Engineering Environment) [69] is a distributed computer-aided software engineering environment that runs on Apollo workstations. It is thought to be one of the most sophisticated configuration tools based on Unix. It distinguishes

between a system model and a configuration thread. The system model describes the components of a software system, and the configuration thread describes the versions of the building blocks (i.e. compilation units) of a corresponding system model. The components of DSEE are:

- The *History Manager* controls source code and provides complete histories of versions.
- The *Configuration Manager* detects the need to rebuild system components and performs the builds when necessary.
- The *Release Manager* saves "good" configurations and helps relate released software to the sources which built the configuration.
- The *Task Manager* relates source code changes made throughout the network to particular high-level activities.
- The *Monitor Manager* watches user-defined dependencies and alerts users when such dependencies are triggered.
- The *Advice Manager* holds general project related information and provides templates for re-doing common tasks.

A drawback of DSEE is that it does not provide general rule for processing configurations [115].

2.4 Characteristics of SCM Techniques

As development of the new test support management system is based on RCS, this section illustrates the functions provided in RCS. However, the general features of SCM tools are necessarily summarised first.

2.4.1 Technical Features of SCM Tools

The role of SCM is to control the evolution of program families. Control is required at two levels, namely the individual component level and the configuration level [64].

At the component level, the change control process is driven by change proposals. Changes are made to individual components via a *check-out/ edit /check-in* cycle. *Access controls* prevent unauthorised changes from being made. *Merging mechanisms* are provided to deal with parallel development, and efficient storage of the resultant versions is achieved through *delta techniques.*

At the configuration control level, Programming-in-the-large techniques (e.g. *module interconnection languages* [23], and *configuration languages* [123]) are being utilised to express both the construction and evolution of system configurations. *Generic representation* methods of system configurations and *version selection mechanisms* for specific configurations are both active areas of research. Additionally the need to maximise productivity has resulted in mechanisms for more efficient rebuild strategies such as *smart recompilation* [114], *opportunistic processing* [62], *parallel and concurrent building* [70].

The underlying *object base* of SCM systems has been advanced to store the system configurations, their constituent components and associated information. These include extensions to the underlying file structure of the operating system [116], and the use of relational and customised databases [121]. Additional research is active in the areas of object-oriented and entity-relationship approaches to object storage [62].

2.4.2 RCS Functionalities

The Revision Control System (RCS) manages multiple revisions of text files. It greatly increases software team productivity by providing the following functions [113]:

1. It stores and retrieves multiple revisions of software objects. It allows the storage and use of one or more releases whilst the next release is under development, with a minimum of space overhead. Changes never destroy the original – previous revisions remain accessible.

2. It maintains a complete history of changes. Therefore the information about how a module which has been modified can be easily and quickly found out.

3. It manages multiple lines of the development.

4. It can merge multiple lines of development. Thus, when several parallel lines of development must be integrated into one main line of development, the merging of changes may be made semi-automatically.

5. It flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.

6. It resolves access conflicts. If two or more programmers wish to modify the same revision, it alerts them and provide a mechanism to ensure that one change will not wipe out another.

7. It provides high-level retrieval functions. Revisions can be retrieved according to revision numbers, symbolic names, dates, authors and states.

8. In conjunction with Make, it provides release and configuration control. Revisions can be marked as released, stable, or experimental. Configurations of modules can be described simply and directly.

9. It performs automatic identification of modules with name, revision number, creation time, author, so that it can help to determine which revisions of which modules make up a given configuration.

There has no RCS mechanism been found which prevents released software items from being casually modified. RCS does not manage non-ASCII files.

2.5 Summary of SCM and Project Requirements

2.5.1 SCM Highlights

SCM is an important software engineering discipline whose application is vital to the development and maintenance of the software systems. Apart from several well-known SCM tools presented in this chapter, there are also quite a number of other tools which have been produced in recent years to aid SCM [124].

Based on the descriptions in this chapter, the advantages of a SCM tool can be summarised as:

- it provides an approach to identifying both objects and their relationships.
- it is a mechanism for controlling and managing the changes to a software object or configuration.
- for a large software development project, it can be used to coordinate the staff working in teams, and improve productivity by reducing or eliminating the confusions caused by the communications among the team members.
- it provides traceability of the status of each configuration object.
- it provides the method to ensure compliance of software deliverables to their required configuration.

Although many tools have been developed to advance SCM, each of them has its limitations. The remaining problems associated with SCM can be categorised as below:

1. The problem with long-term maintainability of the software system [95]:

   The emphasis of SCM has been primarily focused on the code rather than providing: a well-developed baseline mechanism; and traceability and consistency checking

of the code to the system design architecture. This results much of the system requirements and design information being lost during a transition to the maintenance phase.

2. The problem with programming-in-the-large [124]:

   Most existing SCM systems are oriented towards programming languages that do not support programming-in-the-large. Hence, these systems are not often directly applicable to modern programming languages like Ada, Modula-2, C++ and Pascal-XT.

3. The problem with non-textual(non-ASCII) representations [117]:

   There are few SCM techniques which are able to manage the non-textual software representations. The non-textual representations include object code, load images, database and graphics.

4. The problem with distributed and heterogeneous systems [17]:

   Large distributed development teams are usually connected by a net with servers and client machine, and the development environment may be heterogeneous(e.g. different machines, operating systems, software packages, and derived files). Thus, there is a general access and update problem in such system environment. Existing SCM techniques have the problem in dealing with network applications, and few SCM tools have been provided for supporting the distributed and heterogeneous systems.

2.5.2 Requirements Of the Research Project

The research project described in this dissertation is aimed at providing a support environment for the management of software testing(SEMST), which uses SCM techniques to manage test data produced during the testing process. In the past, little research attention has been given to solve the same problem. The motivation for doing this research is

supported by the following:

1. *There is a need for computer-based management of test data produced during the testing process* [section 4.3.2].

   Testing activities are performed in each phase of the software life cycle. A large amount of test data are produced during the testing process. The following is three essential features of the test data components:

   - There are many types of the test data components;
   - These components are subjected to frequent change, resulting in a number of versions;
   - One data component has close relationships with others.

   It would facilitate the testing process if the above three aspects could be dealt with by a computer-aided database management system.

2. *There is a need for computer-based control of retest process during software maintenance* [section 3.6, 4.1.6 & 4.3.3].

   Software maintenance is thought of as the most expensive phase in the software life cycle. Much of the time spent on maintenance is the modification and retest of the delivered software system. Giving automatic support to the maintenance activities will help to reduce the cost of maintenance. Generally, the activities involved in the retest process are the following two categories:

   - determine and select the test cases to rerun.
   - update the old test plan.

   For the first activity, it would be helpful if a way could be provided to tell the testers about which part of test cases has been affected by the modifications of specification or code. From the knowledge of the affected test cases, the testers would be able to determine which part of test cases should be selected to rerun.

After the test cases have been reselected for retest, the old test plan must be changed for use in the next retest. Therefore, help in updating the test plan should be also considered.

3. *It is believed that automatic tools benefit software production in two main senses: lower cost and more reliability.*

2.5.3 Functional Requirements For SEMST

From the above considerations, SEMST is expected to satisfy the following functional requirements:

1. It should supply essential SCM functionalities to manage test data, which include abilities to:
   - load test data;
   - maintain data versions and releases;
   - retrieve and update of any version of the data;
   - manage relationships between the data;
   - trace between the data; and
   - prevent simultaneous update by multiple people.
2. It should provide the information about the changes undertaken to the data.
3. It should be able to identify the change effect on the data and relationships.
4. It should support the management of all test data used at different test levels, including unit testing, integration testing, system testing and regression testing.
5. It should provide an integrated environment to enable the communication of SEMST with a static program analyser, a test case generator and a regression testing tool.

6. It should be a part of a software testing support environment, which can be a component of a software engineering environment or a maintenance environment.

On the whole, SEMST is intended to manage the testing process with SCM techniques support and to provide traceability between the test data across the software life cycle. Section 5.8 presents an analysis of the SEMST requirements in more detail.

2.5.4 Environmental Requirements For SEMST

Hardware

The project has chosen the Unix operating system as the environment because of its good programming utilities. The actual implementation was on a Sun workstation.

Software

The SEMST system is written in C in the Unix environment. Other software packages used in the system development include: C complier, Make and RCS.

As mentioned previously, Make and RCS are of the SCM tools which are available in Unix. Make is used in the project because it helps to maintain all files produced during the system development, and to keep track of the most recent source file versions. After the source file have been changed, Make will regenerate the object code without recompiling unchanged source files. Make also controls the relationships between files and commands.

RCS is chosen as a SCM tool, based on which SCM mechanisms of SEMST are implemented. The major functionalities of RCS have been described in section 2.4.2. The project adopts an existing SCM tool rather than developing a SCM tool by itself. This is because:

- RCS provides the functionalities which are compatible with the project requirements;
- RCS is generally regarded as superior to SCCS [112];
- RCS has interfaces with C and Make;
- The use of existing software tools saves time and resources; and
- The project was planned to develop a prototype environment as an initial product.

Chapter 3

Software Testing

Introduction

Software testing in this dissertation is viewed as the continuous activity and task of planning, designing and constructing tests, and of using those tests to assess and evaluate the quality of work performed at each stage of the software development life cycle.

This chapter presents a study of the software testing discipline. It describes and compares various testing techniques based on the classification of testing techniques into specification-based and program-based strategies. The methods of test case generation associated with these two strategies are discussed. It also presents the descriptions of the testing techniques used in software maintenance. The principles and methods of testing are addressed first in the chapter, and these provide a foundation for the testing techniques to be discussed in the later sections.

3.1 Testing Principles

A testing principle, as used here, means an accepted or professed truth of software testing. There are a few basic testing principles established by testing researchers and practitioners. The common agreed testing principles are:

1. Complete testing is impossible, testing can not guarantee the absence of error in a program [81] [50].
2. Testing work is creative and difficult [81] [50].
3. Testing must be planned [50].

3.2 Testing Methods

The traditional view of software testing is primarily focussed on the code, that is known as *program testing*. With this view, two major forms of systematic method are suggested to perform testing: *top down* and *bottom-up*[78] [81]. In a top-down method, the highest level modules are examined first, and then the process continues building increasingly detailed tests until all of the elements have been tested. The bottom-up method begins with the lowest level modules and continues upward through the hierarchy until adequate tests of the whole system have been completed. Bottom-up corresponds to "build with proven components", and top-down to the strict "hierarchical decomposition" technique.

When testing is focussed on a module, the smallest unit of software, it is called *unit testing*. A program testing procedure starts with unit testing and progresses into *integration testing*, where the focus is on design and construction of the software architecture. The final step of program testing is *system testing*, where the software and other system

elements are tested as a whole [81] [92]. Figure 2 shows a testing procedure.

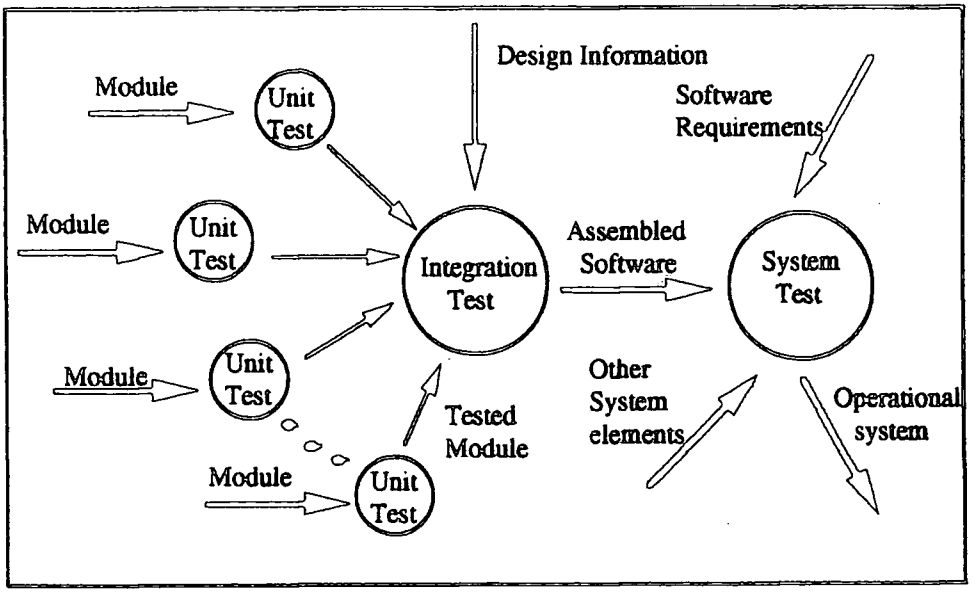

Figure 2. Testing Procedure

In recent years, a different view of testing has emerged, which considers testing not as a phase or step in the development cycle, but as a continuous activity over the entire software development period [86] [50]. In this view, testing is an activity to be performed in parallel with the system development and consists of its own phases. A *test life cycle* was discussed in [50] as a model of representing testing activities embedded within the overall software life cycle. [50] states that a test life cycle includes the following phases:

- **Analysis**: planning and setting test objectives and test requirements;
- **Design**: specifying the tests to be developed;
- **Implementation**: constructing or acquiring the test procedures and cases;
- **Execution**: running and rerunning the tests;
- **Maintenance**: saving and updating the tests as the software changes.

In addition, the *STEP*, Systematic Test and Evaluation Process, developed by Software Quality Engineering, based on ANSI testing standards, was also presented [50] as an example of a testing method.

The evolution of a testing method can be seen by contrasting it with the evolution of a software development method. Software development has gradually been recognised to include analysis, design and other phases, but initially the emphasis of software development was solely on coding.

3.3 Classification of Testing Techniques

Since the application of systematic testing technique requires automated help, much research effort has been devoted to providing automatic (computer-based) tools and techniques to aid software testing. This section concentrates on addressing the classifications of these tools and techniques. Meanwhile, "human testing"(non-computer-based) techniques are worthy of mention because they are still considered as practical and effective ways of finding errors. A *Review* is a widely accepted testing method which can be used throughout the software development process. It is a useful technique when testing requirements and specifications. Many present texts on testing still include the emphasis on a review as well as other "human testing" methods such as *inspection* and *walkthrough* methods [81] [92]. The review technique is discussed in section 4.1.2.

The range of techniques employed in testing is very broad. Generally, these can be classified by two strategic dimensions: *specification-based/program-based* and *static/dynamic.*

Specification-based testing has been termed a functional or black-box approach as it treats the program as a black-box and tests the program according to the functions described in the specification without looking at structural details of the program. In this case, test cases are derived solely from the specification. Program-based testing, sometimes

referred to as structural or white-box testing, is the technique of testing programs based upon the details of program structure. The test cases used for this strategy are derived from the program to be tested. These two testing strategies are described in the next section.

A testing technique that does not involve the execution of the program with data is known as static analysis. It includes *program proving, symbolic evaluation* and *anomaly analysis.* Dynamic analysis requires that the program be executed. It can act as a bridge between specification-based and program-based testing [19]. Both specification-based and program-based testing can be performed either statically or dynamically.

In addition, there is a family of related tools, that neither perform direct tests nor use any specific testing technique. Such tools are considered as test support tools. Within this group, the tools provide their support for the testing activities in a variety of ways. Some tools perform the function of test execution coordination, rerunning test cases for a modified program (regression testing), or comparing the resulting output. Some tools provide a controlled environment in which testing can take place, such as *RUTE – a real-time Unix simulator for PSOS* [75]. The test support systems are discussed in section 5.6. The research described in this dissertation is aimed at developing a tool belonging to this category.

3.4 Specification-Based versus Program-Based Testing

This section is used to discuss two popular testing strategies: *specification-based* and *program-based testing.* In fact, each strategy is associated with methods of test case generation because no program can be tested without selected test cases. The next section will address the techniques for generating test cases.

3.4.1 Specification-Based Testing

Specification-based testing aims to check whether or not the code is complete with respect to the specification. It involves two main steps:

1. Identify the functions which can be tested independently from the specification and partition the program input domain for each identified functional unit into a finite number of equivalence classes.
2. Select the representative elements from each class as the test cases which check whether these functions are performed by the program.

Testing is carried out by executing the code which corresponds to the identified functions. No consideration is given to how the program performs the functions.

Specification-Based testing is essentially the traditional approach to testing a program. However it has the major difficulty of identifying the functions to be tested [51]. It would be considerably more convenient if there were some automated ways of recognising functions and to determine if they had been adequately tested.

The traditional approaches to performing specification-based testing are *equivalence partitioning*, *boundary-value analyzing*, *cause-effect graphing*, and *error-guessing*, which were described by Myers in [81]. Earlier than Myers, Goodenough and Gerhart proposed a method to derive a *condition table* using multiple sources of information where a column in the condition table represents a test case, which is a combination of conditions to be tested [44]. Later on, Weyuker and Ostrand proposed *revealing subdomains* constructed by subdividing path domains based on likely errors, which may be derived from the specification [118]. Richardson and Clarke proposed the *partition analysis* method, which develops a partition by overlaying a program-based partition and specification-based partition [97]. Howden's functional testing employs specification and design information for functional decomposition and applies guidelines for using different functional classes to select test

cases [51]. A more radical solution perhaps is to use *formal specification* methods by which the functions can be completely and clearly specified in formal specification language so that the automated approaches as described in a program-based approach can become applicable. A method of extending program-based techniques, known as *error-based* and *fault-based*, to be applicable to formal specification languages has also been described [98].

Specification-Based testing also depends on the availability of an *oracle*. An oracle is an external source of information about functions, which specifies precisely what the outcome of a program execution will be for a particular test case [51].

More emphasis has been given to the specification-based approach since 1980. Gourly provides a mathematical framework for testing that confirms the need for specification-based testing [43]. Laski illustrates that informal specification does not help uncover errors [68]. There are also some techniques directed toward testing the specification rather than the program. These techniques provide the capability to test the system under development before implementation is underway. For instance, Kemmerer proposes two methods of testing functional specifications based on *InaJo*: symbolic execution of the specification and rapid prototyping by transformation to a procedural form [63].

In spite of many methods proposed, specification-based testing often involves the document reading activities.

3.4.2 Program-Based Testing

If the testing strategy is based on deriving test case from the structure of the program, it is known as program-based testing. The aim of program-based testing is to exercise the program with a certain degree of thoroughness. It may involve the execution of a single path through the program, or it may involve a particular level of coverage such as 100% of all statements have been executed. Over the past years, the notion of *a minimally-*

thorough test (e.g. using the minimum amount of tests to ensure a degree of reliability as high as possible), has occupied researchers. Some of the coverage criteria are given below [81]:

- All statements in the program should be executed at least once.
- All branches in the program should be executed at least once.
- All linear code sequences and jumps in the program should be executed at least once.
- All paths in the program should be executed at least once.

The best testing method is supposed to be an exhaustive one where all possible paths through the program are tested so that the program can be said to be completely tested. However there are five flaws in this approach [81] [19].

1. The number of possible paths in a program is often too large to be tested completely. Because the number of paths is determined by the numbers of conditions and loops in the program, even trivial programs contain a large number of paths.
2. There may be some infeasible paths in the program which cannot be tested.
3. An exhaustive path testing cannot guarantee that the program matches its specification.
4. A program may be incorrect because of missing paths, but exhaustive path testing can not detect the absence of necessary paths.
5. An exhaustive path testing might not uncover data-sensitivity errors.

Program-based testing involves a wide range of program analysis techniques. The main technique is *path selection*, augmented by a test case selection technique. Path selection techniques are concerned with which statements or combination of statements should be

executed. For the path selection problem, a number of path selection criteria have been proposed, such as *control flow coverage*, *data flow coverage* and *perturbation testing*. The following illustrates two common criteria used in path selection.

- *Control Flow Coverage* is the most common path selection criterion. It uses a flow graph (or program graph) to depict logical control flow of the program so that the tester is able to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basic set of execution paths. Test cases derived to exercise the basic set are guaranteed to execute every statement in the program at least one time during testing [92]. *Path coverage* is believed to be the best criterion for path selection. Since it implies the selection of all feasible paths through a program, attaining path coverage is usually impractical. It is generally agreed that branch coverage should be a minimum criterion for path selection. However, achieving this level of coverage is not always straightforward. Statically generating a list of paths that satisfy this criterion usually results in a number of infeasible paths being selected [96].

- *Data Flow Coverage* techniques entail exercising a set of paths that cover particular uses of defined variables. Rapps and Weyuker define a family of criteria for selecting some or all subpaths from a definition statement to some or all uses of that definition statement in the program [94]. Ntafos' criteria requires variable-length chains of alternating definition statements and use statements [83]. The criteria proposed by Laski and Korel requires the selection of different combinations of definitions that reach a statement, where many variables may be referenced [67]. Data flow techniques that attempt to generate only feasible paths by excluding inconsistent pairs of branch predicates are impossible to complete [96].

Recently many researchers on path selection and test data selection techniques have based their analysis on the information provided by *symbolic evaluation* [96]. Symbolic evaluation provides a functional representation of the paths in a program. To create this

representation, it assigns symbolic names for the input values and evaluates a path by interpreting the statements on the path in terms of these symbolic names. The branch predicates for the conditional statements on a path are represented by constraints in terms of symbolic names. After symbolically evaluating a path, its functional representation consists of two parts, *path computation* and *path condition*. The path computation is a vector of algebraic expressions for the output values, which include written output values as well as output parameters and exported global values. The path condition is the conjunction of the path's branch predicate constraints. For the path selection aspects of testing, symbolic evaluation is useful in determining path feasibility for control flow and data flow criteria. It is also being used in the analysis employed by perturbation testing, *a vector space analysis technique* developed by Zeil [127].

Symbolic evaluation is a promising testing technique, which can be used to aid automated test case generation, program proving as well as specification-based testing strategy. However at present stage, symbolic evaluation has several unsolved problems, such as evaluation of loops, module calls and arrays in a program.

3.4.3 Summary

For many years program-based testing was the most common testing strategy and it received much attention in software testing research and development. Its popularity is mainly due to its simplicity and the availability of software tools. In recent years, program-based testing techniques have been criticised for their weakness by focusing only on actual behaviour. There has been a growing literature in testing which claims that specification-based testing techniques should be used to augment program-based testing to enable the testing of intended behaviour as well as the actual functionality of the program. However, although some specification-based testing strategies have been proposed for the specifications written in informal and formal languages, few useful tools have been implemented to support this strategy. This is due to the difficulties of specification-based testing with

automated support.

On the whole, it has been agreed that specification-based and program-based testing are two complementary approaches to software testing. One of them cannot be used to replace the other.

3.5 Test Case Generation

Based on the concepts of specification-based and program-based testing, the possible techniques for generating test case are discussed in this section. Due to the fact that "complete" testing is impossible, the generation of effective test cases is extremely important. The principle of test case generation is therefore to produce a subset of all possible test cases which has the highest probability of reducing the incompleteness as much as possible. Basically, there are two types of test cases used in the normal testing process, known as *functional test cases* and *structural test cases.* Functional test cases are used to test all of the functions of a software system. Functional test cases should also be used to test boundary conditions, special cases and error handlers in a software system. Functional test cases can be derived from the system requirements, specifications, design information, or from the code itself [85]. In this section, the discussion is only focusing on the specification-based approaches to generating functional test cases. Structural test cases relate directly to the program's structure, logic, control flow and data flow. These can be derived solely from the code.

3.5.1 Specification-Based Approaches

The goal of specification-based testing of a software system is to find inconsistencies between the actual behaviour of the implemented system's functions and the required be-

haviour as described in the system's functional specification. To achieve this goal, it requires that

1. the test cases be executed for all of the system's functions;
2. the test cases be designed to maximise the chances of finding errors in the software system.

A standard approach to generating specification-based functional test cases is first to partition the input domain of a program into a finite number of equivalence classes, such that all elements within an equivalence class are essentially the same for the purpose of testing. The next step is to select test cases from each class of the partition. If the main emphasis of the testing is to attempt to show the presence of errors, then the assumption is that any element of a class will explore the errors. If the main emphasis of the testing is to attempt to give confidence in the correctness of the software, then the assumption is that correct results for a single element in a class will provide confidence that all elements in the class would be processed correctly [85].

On the use of specification-based approach to generate test cases, there have been various methods developed over the past years, quite a few of which describe the techniques for creating test partitions. However, the partitioning process lacks a systematic approach [85]. The following paragraphs are used to describe some well-known methods proposed for generating functional test cases by this approach.

- *The Condition Table Method* was proposed by Goodenough and Gerhart [44]. It is used to construct a condition table in which each column represents a combination of conditions that can occur during the execution of a program. By examining the program's specifications, the conditions that have a significant impact on the execution behaviour of the program are identified.

- *Cause-Effect Graphing* is the strategy originally defined by Elmendorf [29] and illustrated by Myers [81]. It begins with the identification of each individual function or command of the system to be tested. Then for each function, all significant causes that influence the function's behaviour and all effects of the function are identified. The next step is to construct a graph that relates combinations of the causes to the effects they produce. Test cases are defined for each effect by considering all combination of causes that produce that effect. Although the use of this method can produce effective tests, the method is difficult to apply in practice. In particular the cause-effect graph can become very complex when a function has a large number of causes.

- *The Revealing Subdomains* method is proposed by Weyuker and Ostrand [118], which combines the specification-based and program-based approaches to derive a partition of a function's input domain into revealing subdomains. A revealing subdomain contains elements that are either all processed correctly or all processed incorrectly. Once such a subdomain has been identified, executing the program on a single element is sufficient to test the entire subdomain. The revealing subdomains can be constructed by identifying the most likely places for errors to occur. The first step is to create a *problem partition* from the specification, by looking for classes of inputs that should be treated the same way by the program. The next step is to create a *path partition*, whose equivalence classes contain inputs that actually are treated the same way by the program. The partition used for specification-based testing is then created by intersecting the problem partition and the path partition, creating a set of equivalence classes whose elements both should be and are treated the same way by the program. A test set is built by choosing one element from each of the testing partition's classes.

- *Equivalence Partitioning* is the method proposed by Richardson and Clarke [97] to generate functional tests based on both specification and program. It is similar to the revealing subdomain approach in the way that it partitions a function's input domain into a *procedure partition*, which is the intersection of a program-based

path domain and a specification-based *specification domain.* The path domain is constructed by applying symbolic execution to the program. To construct the specification domain, the authors assume that the specification is presented in a formal specification language, to which symbolic execution techniques can be applied. Test data are selected according to the types of errors to be detected.

- *The Category-Partition Method* is proposed by Ostrand and Bacler[85]. The method analyses the specification and identifies testable functional units, categorises each function's inputs, and then partition categories into equivalence classes. The main characteristics of this approach are: test specification can be easily modified when necessary and it can control the complexity and number of tests by annotating the tests specification with constraints.

Despite the common agreement on this key approach to generating test cases and its progress since the 1980's, none of these test case generation approaches has been sufficiently well-defined to be generally applicable. Nevertheless, this research represents a significant step in the achievement of reliable software since it allows earlier derivation of tests and an oracle which clearly determines whether or not the output produced is correct.

3.5.2 Program-Based Approaches

For program-based approaches, test cases are derived from the program according to certain test criterion. The well-known test criteria are *statement coverage, branch coverage* and *path coverage.* In particular, great attention has recently been paid to path coverage criterion. Testing based on this criterion is called *path testing* which is intended to execute all paths reaching from an entry to an exit on a control flow graph of a program. It has been realised that complete testing of all paths is in general impractical. Therefore a technique called *sensitising the path* [4] has been proposed in order to make path testing practical. In this technique a subset of paths is selected, and test cases that will cause

those paths to be executed are found. Unfortunately, path testing has two major problems. One problem is that most path selection techniques proposed do not provide the guidelines for selecting test data. Another problem is that even exhaustive testing of all paths in the program may not necessarily find all errors [section 3.4.2].

The best known approach to deriving test cases for path testing is achieved by the technique of *symbolic evaluation*, which involves four steps:

1. To construct a program flow graph. The program is preprocessed to create a digraph representation of control flow in the program. Other relevant information is collected for later analysis.

2. To select the paths. The path selection is concerned with selecting program paths that satisfy a level of test coverage. The process can be manual or automatic, static or dynamic. In automatic static selection, paths are automatically selected by symbolic execution.

3. To execute program symbolically. Once a path is selected, symbolic execution is used to generate path constraints. Input data satisfying these constraints will result in the execution of that path.

4. To generate test cases. This step includes selecting data that will cause the execution of selected paths. A widely used technique is linear programming algorithms by which numerical solutions to the inequalities of path constraints can be found.

The *Domain testing* technique [15] [119] appears to be promising for a large class of data processing programs. The method is also a path-oriented testing approach. It concentrates on the detection of domain errors by analysing the path domains and selecting test data "on" and slightly "off" the closed borders of each path domain. If the correct results are produced for each of the on and off test points, the border must be "close" to the correct border. An undetected border shift can only occur if the on test points and the off test

points lie on opposite sides of the correct border. The undetectable border shifts are kept "small" by choosing the off testing points as close to the border being tested as possible. With the proper selection of on and off test points, a quantified error bound measuring the set of elements placed in the wrong domain by an undetected border shift can be provided.

In [98], program-based test case generation techniques are classified as *error-based* and *fault-based.*

Error-based techniques are aimed at revealing specific types of errors, where an error is a mental mistake by a programmer or designer. Symbolic evaluation and the domain testing strategy are considered to be in the range of error-based techniques. Error-based strategies are sometimes referred to as *error-sensitive heuristics.*

Fault-based testing selects test cases that focus on detecting particular types of faults, where a fault is a mistake in the source code. It consists of "rules" that are applied to the source code to select test data sensitive to commonly-introduced faults. The *RELAY* model [99] provides a fault-based technique for test case selection. *RELAY* guarantees the detection of errors caused by any fault in a user-chosen fault classification. The *RELAY* model proposes the selection of test data that originates an error (introduces an incorrect state) for a potential fault of some type and transfers that error along some route through computation and data flow until a failure is revealed. *RELAY* develops revealing conditions that describe how to distinguish the source from the variant. Any test data set satisfying the revealing conditions contains some test datum that reveals the chosen fault. *RELAY* is limited to the detection of errors resulting from a single fault in a module.

3.6 Testing During the Maintenance Phase

Software maintenance can account for over 60% of all effort expended in a software life cycle [92]. It falls into four categories: *adaptive, perfective, corrective* and *preventive*

maintenance. Each of these terms may best be described by their definition, which are stated below [55]:

1. Adaptive Maintenance – the maintenance performed in order to make a software product usable in a changed environment.
2. Perfective Maintenance – the maintenance performed to improve performance, maintainability, or other software attributes.
3. Corrective Maintenance – the maintenance performed specifically to overcome existing faults.
4. Preventive Maintenance – the modification of a system to ease future maintenance.

Modification to the software after its completion is inevitable: the software system and its application will evolve as it is adapted to a changing environment, changing needs, new concepts and new technologies; a software system will have an increasing number of functions, components and interfaces; old modules may be expanded for uses beyond their original design. Thus, much of the time spent on software maintenance is in modifying and retesting the software.

A widely used testing technique in software maintenance is called *regression testing.* It is well recognised that many of the errors appearing in a software product do not exist in the original implementation, but are accidentally introduced during a number of modifications undertaken in subsequent revisions. In order to combat such problems, the original implementation of the software product should include a thorough set of test cases or procedures that exercise and verify all functional aspects of the program. It should possess the capabilities of retaining and extending these test procedure during the software life cycle. Thus, in subsequence error corrections or modifications, all or at least a specific subset, of the previously executed test cases can be rerun in order to ensure that the changes had only the local effects intended. Regression testing is the technique

used to solve the above problem and manage the process of software revalidation after modification to the software. Efficient and effective regression testing can reduce the cost of maintenance.

Regression testing has been identified as two types [71]: *corrective regression testing* and *progressive regression testing.* Corrective regression testing involves a constant specification and progressive regression testing is concerned with a modified specification.

There are two major problems associated with regression testing[58]: the *test selection problem* and the *test plan update problem.* The first problem is concerned with the selection of test cases from the existing test plan and generation of new test cases for the modified program or specification. The second problem involves the management of the test plan.

The earliest regression testing tool known as the Automatic Test Unit System (AUT) was developed by IBM in 1972 [47]. It was mainly used in unit testing. Subsequently, in 1975, the General Electric Research and Development Centre implemented an automatic software test driver, which became known as the Test Procedure Language(TPL/F) system. These early regression testing tools have failed in gaining wide acceptance because they were restricted to use a specific language [47].

Regression testing is currently receiving more research attention, but most methods are restricted to testing at the unit level.

3.7 Other Testing Techniques

Apart from the software testing techniques addressed above, there are also a number of other methods which have been developed to aid software testing.

- *Program mutation* [12] is an error-based technique for the measurement of test data adequacy. Test adequacy refers to the ability of the data to ensure that certain errors are not present in the program under test. In mutation testing, test data is applied to the program being tested and its "mutants"(i.e., programs that contain one or more likely errors). If a program passes a mutation test, then either the program is correct or it does not contain a most likely error. A difficulty for mutation testing is to determine the equivalence of mutant program to original program. When a mutant program is equivalent to the original program, mutation testing may not provide correct results. Another problem with mutation testing is that a large number of mutants can be generated for even a simple program.

- *Random testing* [28] is essentially a black-box testing strategy in which a program is tested by randomly choosing a subset of all possible input values. The distribution may be arbitrary, or may attempt to accurately reflect the distribution of inputs in the application environment. Random testing is usually considered as very weak testing technique. The problem is that it may seem that there is no guarantee to complete coverage of the program. However, Duran and Ntafos [28] claim that random testing is a useful validation, and they present the some results of actual random testing experiments. A recent study by Loo and Tsai [74] shows that random testing works well on several kinds of programs, but not at all. They provide the conditions under which random testing can be effective:

  1. When the program being tested is error-prone. For example, a program in an early stage of its development.
  2. When the expected outputs of the test inputs are known or can be easily obtained. This implies that a lot of test cases can be generated with a low cost.

  Without the above conditions the effectiveness of random testing is significantly reduced, because test case generation will be expensive and not many faults can be found.

- *Real-time software testing.* Real-time systems possess additional attributes that must be given special consideration in the testing process. The typical attributes of real-time software are the large number of modules that have to be integrated and tested and the same sequence of test cases, when input at slightly different times, may produce different outputs. Real-time software testing can be characterised as host and target computer testing. The goal of host computer testing is to reveal errors in the modules of software. Most of the testing techniques that are used for testing on a host computer are the same as for non-real-time applications. In target computer testing, module testing is conducted first. Integration testing is then conducted sometimes using an environment simulator to drive the target computer.

- *Proving program correctness.* Program proving, recently referred to as formal verification involves the use of rigorous, mathematical techniques to demonstrate that a program conforms to its requirement specifications. The method of *inductive assertion* (also called an input-output assertion), developed by Floyd [37] was seminal to the field of formal verification [33] [41]. During the early years, formal verification was concentrated on the program, known as program verification. There had been a number of methods developed on the basis of Floyd's work to prove the correctness of a program. However, several disadvantages of only verifying a program have been recognised[103]:

  - The program may be written in such a way that verification is very difficult (e.g. some implementation-dependent constructs whose semantics are not clear may have been used to satisfy efficiency requirements).
  - The programming languages used may be so low level that verification is impossible (e.g. Assembly language).
  - If the verification of a program uncovers design errors, it may involve considerable work in redesign and reimplementation of the program. It is far better to detect these errors at the design stage.

- Since an implementation is usually larger than a design, program verification is longer and more expensive than design verification.
- If the specification is incorrect, it is impossible to verify the program.

With the growth of the *formal methods* in the area of software engineering, the attention of those working in formal verification has been turning to the specifications written in a mathematical language, namely formal specifications(e.g. VDM [59] and Z [106]). Formal specifications employ mathematical notation in order to achieve both precision and conciseness. The key to the brevity is *abstraction*. When the requirements are specified in such a mathematical form, proving correctness of the requirement specifications can be based on the proof theory which has been well-established in the area of mathematics. The *deduction* and *induction* methods with the relevant inference or reasoning rules are used in mathematics to prove the propositions and predicates. These two proof methods have been introduced in many formal methods texts(e.g. Jones [59]) for proving specifications, as a formal specification is in fact an integration of a set of propositions or predicates. A formal specification can be constructed with the proof of its correctness at the same time. One of the advantages of formal methods is verified design [59]. Verified design uses the concept of proof as a way of checking design steps. Steps in a systematic development can be based on, and verified against, a formal specification. The idea that programs are mathematical texts shows the possibility of reasoning about their formal relationship to specifications, which means that it is possible to transform the design automatically into several different programs, depending on the system required. Thus, by verifying the design, as many errors as possible can be eliminated at an early stage of the software system development, and only a single verification is necessary for the implementation. The advantage of this formal verification approach is its attempt to produce a correct specification. However, such a verification process is still very expensive because both the design and the implementation should be verified.

Newer ideas are to ensure that correct programs are generated from the specifications so that the work on verifying the implementation can be saved. It is viewed that both the specifications and the implementations are all programs, but the former are abstract programs, not necessarily executable, and the latter are executable programs [80]. With this view, the process of transformation of the specifications to the programs is called *refinement.* If the transformation process correctly follows refinement rules and steps, the derived programs are believed to be correct with respect to the specifications. The advantage of this transformational approach compared to formally verifying that a program meets its specification(i.e. the approach mentioned before) is that the distance between each transformation is less than the distance between a specification and a program[104]. Program verification is usually very long and impractical for large-scale systems, but a transformational approach which is made up of a sequence of smaller steps may be more effective. However, this process is not easy to perform. Choosing which transformation to apply is a skilled task and proving the correspondence of transformations is difficult[104].

In practice, a specification seems never initially complete enough to allow the complete system to be generated from it. Few software systems have been developed using refinement and transformation, and it is unlikely that a pure refinement and transformation approach will be adopted for the development of very large system. However, recent research in formal methods has been making this process a more practical one, and the incorporation of this process model into other process model is likely to lead to the improvement of software development process[104].

The opposite process to refinement is abstraction, which is concerned with recovery of a specification from the code. This is useful for program maintenance, especially for maintaining the programs for which no accurate written specification exists.

3.8 Automated Testing Tools

Automated testing tools provide the following attributes that are not as easily attainable by manual testing approaches:

- Improved organisation of testing through automation
- Measurement of testing coverage, and
- Improved reliability

The following presents a number of categories for test tools which have been developed over the past years, according to [22]:

- *Static Analysers.* Static analysers are programs that analyse source code to reveal global aspects of program logic, structural errors, syntactic errors, coding styles, and interface consistency. They consist of a front end language processor, a data base, an error analyser, and a report generator.
- *Dynamic Analysers.* Dynamic analysers include the operation of *coverage analysis, tracing, tuning, simulation, timing, resource utilisation, symbolic execution, assertion checking,* and *constraint evaluation.*
- *Symbolic Evaluator.* Symbolic evaluators are programs that accept symbolic values and execute them according to the expression in which they appear in a program. They are used to support test data generation, assertion checking, path analysis, and detection of data flow anomalies. Some of the well-known systems include SELECT [10], EFFIGY [65], ATTEST [14], DISSECT [52], and SMOTL [7].
- *Test Data Generators.* A test case generator is a tool which assists a user in the generation of test data. The example systems are the ATTEST [14] and SMOTL [7].

- *Program Instrumenters.* Program instrumenters are systems that insert software probes into source code in order to reveal its internal behaviour and performance. The main applications include coverage analysis, assertion checking and detection of data flow anomalies. The PET – *Program Evaluator and Tester* [107] is an example of program instrumentors.

- *Mutation Testing Tools.* An automatic mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. In addition to a mutation"score" that indicates the adequacy of the test data, a mutation system provides an interactive test environment and reporting and debugging operations which are useful for locating and removing errors. FMS.3 [108] is a Fortran mutation analyser.

- *Environment Simulators.* An environment is a specialised automatic system that enables the tester to model the external environment of real-time software and then simulate actual operating conditions dynamically.

There are many tools developed for software testing but have not been used widely to date. The restricted scope of many tools and the difficulty in applying the more powerful tools have limited their utilisation across software engineering application areas. However, recent work in AI-based testing techniques shows promise [92].

It is said that "Testing never ends, it just gets transferred from you (the developer) to your customer. Every time your customer uses the program, a test is being conducted."[92]

3.9 Comparative Review of the Testing Techniques

In section 3.3, a classification of software testing techniques was presented. Manual methods of finding software errors such as *review, inspection* and *walkthrough* have been considered as one type of testing technique in this dissertation, and is called "human testing".

While much research effort has been paid to develop automatic (computer-based) testing techniques, human testing still plays an important role in evaluating the quality of software products produced during the system development process because of its simplicity in practice. Since the dissertation is intended to address the management of the testing process, human testing techniques are worthy of consideration. However, human testing is generally believed to be less cost-effective and reliable than automatic testing, although it is generally agreed that both methods should be employed.

The systematic testing process requires automatic support. Over the past years, a wide range of the tools and techniques have been produced to aid automatic software testing. Basically, these tools and techniques can be seen as two families. The first family consists of the testing techniques used to directly perform the tests, and is primarily addressed in this chapter. The second family is those techniques provided for supporting the tests, called test support techniques, which are discussed in section 5.6.

Various testing techniques can be essentially classified into either specification-based/program-based testing strategies or static/dynamic testing strategies. The methods of generating test cases are associated with each testing strategy. Because the SEMST system, presented in chapter 6, supports the management of test cases derived on the basis of specification-based/program-based strategies, this chapter is mainly devoted to describing the testing techniques developed for specification-based/program-based testing.

Based on the previous sections of this chapter, the major specification-based and program-based testing techniques are reviewed and compared in the following sections in order to stress their distinct advantages and disadvantages.

3.9.1 Specification-Based Testing Techniques

Specification-based testing, also known as functional or black-box is designed to validate functional requirements without regard to the internal workings of the code. The techniques of this approach focus on the information domain of the software, generating test cases by partitioning the input and output domain of the program according to the functions described in specification. Section 3.4.1 and 3.5.1 presented a number of well-known techniques for specification-based testing, which include *the condition table method* [44], *cause-effect graphing* [81], *the revealing subdomains* [118], *equivalence partitioning* [97], and the *category-partition method* [85]. In the following, these techniques are evaluated.

1. The Condition Table Method

   This method was developed by Goodenough and Gerhart [44]. From the descriptions in [44], the main benefit of this method can be said to be that it provides a way to develop and describe the test predicates (i.e. a set of descriptions of conditions and combinations of conditions relevant to the program's correct operation [44]). Because each column in the condition table contains a combination of conditions that can occur during the execution of a program, each column actually represents a test predicate. The test predicates are useful for test case selection. However, [44] did not discuss how to construct the condition table automatically. To identify the conditions, the testers must read the program's specification carefully.

2. Cause-Effect Graphing

   Cause-effect graphing is a traditional specification-based testing approach which has been included in many testing texts [81] [92] [91] [85]. The advantages of cause-effect graphing method can be summarised as [91]:

   - it is a systematic method of selecting a set of the test cases which have a high probability of detecting errors in a program.
   - it provides a way to identify individual function from the requirement specification.

- it has the added capability of pointing out incompleteness and ambiguities in the requirement specification.

The criticisms for this approach can be illustrated as follows [91] [85]:

- it is difficult to apply in practice. When a function has a large number of causes, the cause-effect graph can become too complex to deal with.
- it does not produce all the useful test cases that can be identified.
- it does not adequately explore boundary condition.
- it is difficult to update when a change is required.
- it is difficult to verify its correctness after a change to it.

3. The Revealing Subdomains

Weyuker and Ostrand [118] proposed a technique that attempts to construct revealing subdomains by identifying the most likely places for errors to occur. Two steps to the test cases generation by this method were described previously. The advantage of this technique is that it combines both specification-based and program-based approaches to deriving test cases. The main limitation of this technique is its difficulty in providing formal or systematic guidelines for creating problem partition from the specification.

4. Equivalence Partitioning

This technique, developed by Richardson and Clarke [97] is similar to the revealing subdomain technique because it also uses both specification-based and program-based approaches to generating test cases. However, it has its own characteristics which differ from the revealing subdomain approach.

- The problem of systematically creating the specification domain, existing in the revealing subdomain method, is solved in this method by assuming that the specification is presented in a formal specification language, to which symbolic execution techniques can be applied.

- This method relies on the types of errors for generating test case. There are two types of errors identified in this method: computation errors and domain errors. Unlike this method, the revealing subdomains method identifies the errors which are most likely to occur.

The main drawback of the equivalence partioning method is that it crucially depends on a formal specification to allow the symbolic execution that creates the specification domain, but many specifications today are still written in natural language.

5. Category-Partition Method

   This method was proposed by Ostrand and Bacler in 1988 [85]. It has several merits that the above methods lack:

   - it is applicable to an informal specification. For a system that is specified in natural language, this method could be used to determine an appropriate set of specification domain, by converting the informal representations of these domains into an intermediate representation similar to that produced by symbolic execution of a formal specification.
   - it provides the tester with a systematic method for decomposing a functional specification into test specifications for individual functions.
   - it allows the tester to modify the test specification.
   - it can control the complexity and number of the tests by annotating the tests specification with constraints.
   - it emphasizes both the specification coverage and error detection of testing.

   Unfortunately, this method is not completely automatic. It involves testers in document reading activities [98].

To summarise, a number of specification-based testing techniques have been proposed and research in this area has been in progress since the 1980's. However few automatic tools have been implemented to support this strategy. The major difficulty of this approach

is the identification of the functions to be tested from the specification. Specification-based testing requires knowledge of the specification, and attaining automatic support for such approach needs formal specification, on the basis of which program-based testing approach can be employed. Gourlay [43] has used mathematical theory to assess the above specification-based testing techniques. Because of the difficulty in practice, none of these testing techniques has been experimentally evaluated on their effectiveness.

3.9.2 Program-Based Testing Techniques

Program-based testing is also termed structural or white-box testing and focuses on the program's structural details. In this approach, test cases are derived from the program, and are used to assure that the program is exercised with a certain degree of thoroughness. The major techniques used in program-based testing are: *path testing*, *statement testing*, *branch testing* [81], *symbolic evaluation* [96], and *domain testing* [119], and *data flow testing* [94], which were discussed in the previous sections. Except data flow testing, other testing techniques listed above are path-oriented and based on the use of control flow of the program [120]. This section reviews these techniques with an assessment.

1. Path Testing

   Path testing requires that all possible paths in a program be executed at least once over the selected test cases [81]. It is considered as the strongest test coverage criterion for test case generation. However, since even a small program can contain a large number (potentially infinite) of paths, complete path testing is impractical. Section 3.4.2 illustrates five flaws associated with this approach.

2. Statement Testing and Branch Testing

   Because complete testing of all paths in a program is in general impractical, statement and branch testing criteria are used in order to achieve a minimal set of test coverage. Statement testing means that all statements in the program should be

executed at least once. This approach is generally regarded as the weakest test criterion in program-based testing as it fails to detect many kinds of errors [81]. Branch testing requires all branches in the program be executed at least once, that is each predicate decision assumes a true and a false outcome at least once during the test execution. This technique is generally considered to be a minimal testing requirement. Branch testing is stronger than statement testing because branch testing implies statement testing. However, it is still shown to be inadequate [81].

It is said that about 65% of all bugs can be caught in unit testing, which is dominated by path-testing methods(50% - 60% of all bugs caught), of which statement and branch testing dominate [4]. These testing techniques would be more effective when they are combined with other methods, such as limit checks on loops.

3. Symbolic Evaluation

   Symbolic evaluation, sometimes referred to as symbolic execution[96], provides a functional representation of the paths in a program by creating a path computation and a path condition after symbolically evaluating a path. Symbolic evaluation is a promising testing technique, which can be generally used to aid [96]:

   - automated test case generation
   - path selection
   - program proving
   - determining path feasibility
   - partition analysis
   - specification-based testing strategy

   An investigation of the effectiveness of symbolic evaluation and some other testing techniques has been done by Howden [52]. For 28 errors occurring in 6 programs, the reliability[1] of each technique was indicated in his study. The conclusions were that

[1] A testing technique is reliable for an error only if every test data set that satisfies the criterion of that technique is guaranteed to reveal the errors[52].

path testing was reliable for 18 errors of the 28 errors; branch testing was reliable for only 6 errors; symbolic testing of the set of path chosen to approximate path testing guaranteed the detection of 17 errors; and the combination of symbolic evaluation with other testing and analysis methods was reliable for 25 errors of 28 errors.

There are three major problems which have not been well-solved in symbolic evaluation. These problems are the evaluation of loops, module calls and arrays in a program.

4. Domain Testing

Domain testing attempts to uncover errors in a path domain, known as domain errors, by selecting test cases on and near the boundaries of the path domain [119]. The main features of this approach include [15]:

- it provides a formal approach for satisfying the often suggested guideline that boundary conditions should be tested.
- it can be easily modified to handle equality and nonequality predicates.
- it may inadvertently uncover computation errors, since the program is executed on several test points.

Generally, the domain testing strategy requires at most $s(N+3)$ test points per domain, where N is the dimensionality of the input space in which the domain is defined, and s is the number of border segments in the boundary of the specific domain[120]. White and Cohen[119] have addressed that "for linearly domained programs, with each Off point chosen a distance D from the corresponding border, the domain testing strategy is guaranteed to detect all errors of magnitude greater than D using no more than $s(N+3)$ test points per domain, where N indicates the dimensionality of the input space and s is the number of predicates along the path to be tested".

Domain testing has some limitations as it is based on the following assumptions [119]:

- coincidental correctness does not occur for any test cases.

- a missing-path error is not associated with the path being tested.
- each border is produced by a simple predicate.
- the given border is linear.
- the input space is continuous rather than discrete.

The above path-oriented testing techniques all suffer the following two problems:

- unable to deal with coincidental correctness problem
- unable to deal with missing path errors

5. Data Flow Testing

The testing strategy based on data flow analysis is known as data flow testing. A family of test case selection criteria for data flow testing was defined in [94] [83] and [67]. Existing data flow testing techniques include: *all-du paths*(d – defined or initialised, and u – used), *all-use*, *all-use/some-c-use and all-c-use/some-p-use* (c – used in a calculation, and p – used in a predicate), *all-definition*, *all-p-use* and *all-c-use* [4] [58]. Among these, the all-du paths technique is the strongest data flow testing strategy. The advantages of data flow testing can be described as follows [91] [128] [96]:

- it is easier to achieve than path testing. Path testing involves path selection activities which are often difficult in practice, but data flow testing is only based on a program flow graph which is usually easy to construct.
- data flow coverage can be used as one of the path selection criteria.
- data flow criteria are more selective than the control flow criteria with respect to selection of simple transference paths.
- data flow testing bridges the gap between all paths and branch testing.

An experiment which compared random testing, branch testing and data flow testing(au strategy) has given the following result[83]:

Strategy	Mean No. Test Cases	Bugs Found(%)
Random testing	100	79.5
Branch testing	34	85.5
Data flow testing(all use)	84	90.0

There have been relatively more experiments on effectiveness of data flow testing than other testing strategies, a summary of which can be found in [4].

The limitations of data flow testing have been generally considered as [128] [4]:

- it cannot detect unexecutable paths.
- it is weaker than path testing strategy.

To summarise, program-based testing techniques have drawn much research attention over the past years, and these techniques have been practically available. Section 3.8 described some such automated tools developed in the past. Among the program-based testing techniques reviewed above, symbolic evaluation appears to be promising [96] [19]. Symbolic evaluation can be used to assist all of the above testing techniques [96]. Howden [53] found that combining both symbolic evaluation and other testing methods was reliable for more errors than either method used alone. It has been realised that using a variety of testing techniques together produces more reliable software [19]. However, program-based testing has one serious shortcoming, namely that this strategy entirely depends on the internal structure of the program, but the structure itself may be incorrect with respect to the functional requirements.

3.9.3 Regression Testing Techniques

Regression testing concerns the revalidation process in software maintenance. It can also be performed on the basis of specification-based/program-based testing approaches. In comparison with the testing techniques mentioned above, regression testing focuses on detecting the errors which may have been caused by program changes. Two types of regression testing have been identified and they were indicated in section 3.6. Generally, regression testing involves the following major activities:

- identifying the effects of the change to code or to both code and specification;
- selecting the existing test cases and new test cases which will be used to test the affected region;
- executing the modified program based on the selected test cases;
- ensuring that the modified program still performs the intended behaviour specified in the (possibly modified) specification;
- updating the old test plan for the next regression testing process.

There have been several issues on regression testing strategies, which include Fischer's method[36], Yau et al's method[126] and Hartmann et al's method [48]. An evaluation of these regression testing strategies can be found in [58].

Section 3.6 illustrated some regression testing tools developed in early 1970's. However, these tools were limited to use a specific language and test the program at unit level. Hence these tools have gained little wide acceptance.

At the present stage, regression testing is not effective. The following descriptions present the problems associated with current regression testing techniques [48] [47], and these problems are considered by the research project described in this dissertation.

- *test selection problem.* Using an entire baseline set of test cases to validate a few changes may cost a large amount of time and computational resources. It needs a systematic selection of the test cases to ensure a reliable revalidation process.

- *test cases maintenance problem.* Test cases used in regression testing comprise two classes. The first class is the old test cases derived during the system development process, and the second is the new test cases generated during maintenance. This means that all baselined old test cases and new test cases must be maintained in order to support a continuous regression testing activity during software maintenance. Furthermore, regression testing requires the maintenance not only for input test data but also for the resulting test output. The combination of the input tests and the resulting outputs makes a test suite.

None of the regression testing strategies has been fully evaluated. To develop an effective regression testing strategy much research is required.

3.10 Summary

Based on the classification of various testing techniques into specification-based/program-based strategies, this chapter has presented the descriptions of a number of well-known testing techniques associated with these two approaches, together with their individual advantages and disadvantages. To conclude, specification-based testing and program-based testing are two complementary approaches to software testing.

Chapter 4

Software Testing Management

Introduction

This chapter presents an investigation of software testing management and stresses its necessity. The chapter is divided into five secstions: section 4.1 is used to describe the testing activities in the software life cycle; section 4.2 addresses the characteristics of test data components concerned by this research and describes their relationships; section 4.3 discusses the problems in testing so as to indicate the need for testing management; section 4.4 applies SCM methods to testing; and section 4.5 is a summary of the chapter.

4.1 Testing In the Software Life Cycle

The term *software life cycle* is used as a model to represent the process of software project development and maintenance. The traditional *waterfall* model of software life cycle includes five primary phases shown in figure 3. In this model, each phase of system development has identifiable activities and end products(or deliverables). There are also well-defined links between the end products and these serve as the basis for the various testing activities described throughout the dissertation.

The traditional view of software testing is simply a process of exercising code with a number of test cases. This dissertation adopts a wider view of software testing, shown in figure 4: *testing is a broad and continuous activity throughout the software life cycle. It embraces a wide spectrum of activities ranging from informal design reviews, through rigorous test case analysis to formal proofs of correctness.* With this view of software testing, six essential activities, namely early test planning, reviews, unit testing, integration testing, system testing and revalidation are described in the sections that follow.

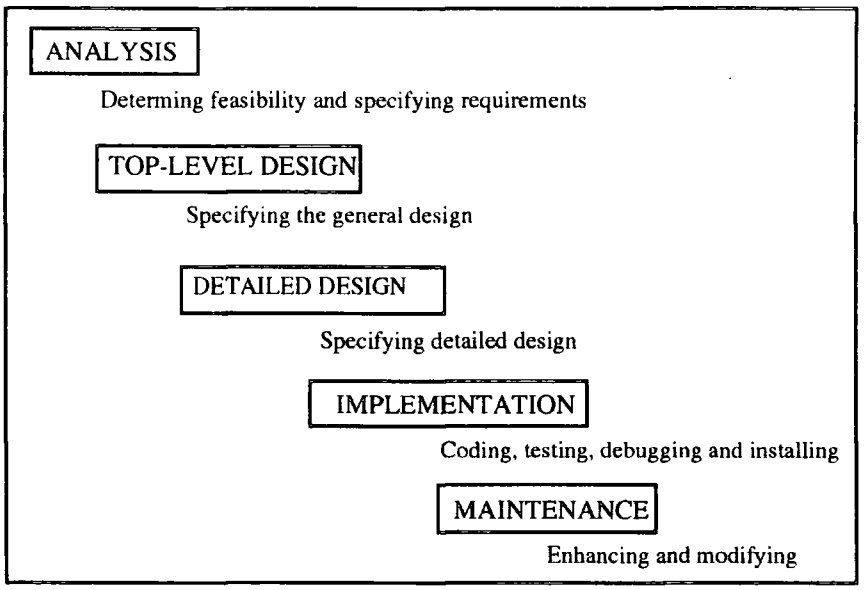

Figure 3. The Software Life Cycle Model

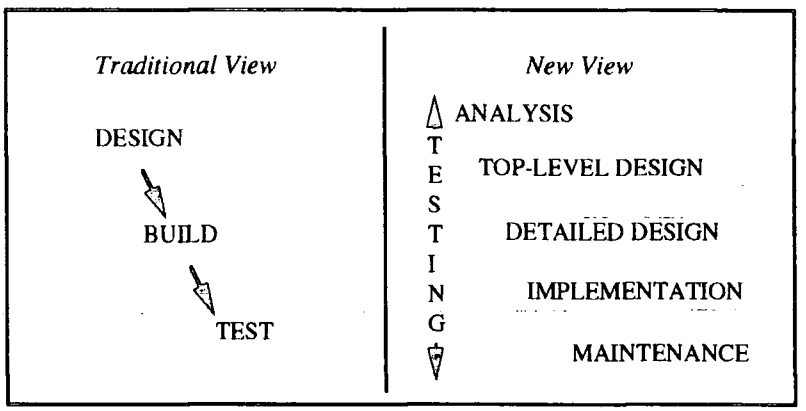

Figure 4. Testing In the Life Cycle

4.1.1 Early Test Planning

Early test planning, as a key factor for project success, is addressed in [32]. During the initial periods of the project, test planning should be emphasised and the plans, schedules and budgets for the implementation of the project should be established. These plans serve as the basis for defining the specific tools, techniques, and methods to be applied to the test implementation. An appropriate system test plan could have built a smooth, controlled flow of data and responsibility, clear, baselined requirements, controls over resources, data, the accomplishment of work, and a sustained project focus toward testing milestones. This pre-planned and controlled application of resources, technical methods, and tools and techniques would have assured that testing structure was consistent with other project areas and was tailored to the characteristics of the project.

The test planning activities include *system test planning, integration test planning* and *unit test planning.* Implementation of the test plans should result in a structured flow of data and responsibility through all levels of test and integration. The basic elements of each plan have been described [32].

The *System Test Plan* defines the requirements of system testing. It describes the test case definition and the environment and methods to be used for qualifying the hardware and software system operational components. Also, the organisational structure, resource requirements, and system technical and management controls to be applied by the program during the system testing are described.

An *Integration Test Plan* describes how the individual software subsystems will be integrated and qualified in an operational configuration. A separate software integration test plan should be developed for each software subsystem in the system configuration.

The *Unit Test Plan* is the basis for the individual unit test specifications included in the Unit Development Folders. It describes unit test organisation, test procedures, test

case definition and test tools and techniques used.

The format of the plans mentioned above has been illustrated by Evans [32]. ANSI/IEEE (Std 1008-1987) [56] provides the standard format for writing a test plan.

4.1.2 Reviews

Reviews in this dissertation are considered as a testing technique, but in some other cases, reviews are described as a software quality assurance activity during testing [92].

A review is performed using a group of people to:

1. point out needed improvements in the end products;
2. confirm those parts of an end product in which improvement is neither desired nor needed.
3. achieve technical work of more uniform quality than can be achieved without reviews, in order to make technical work more manageable.

There was a time when a review was only used to examine the quality of technical work, known as a *technical review.* Now many different types of reviews can be seen throughout the development life cycle. This includes the reviews of requirements documents, design, code logic, test plan and test documentation.

Reviews may be formal or informal. Formal reviews involve the accurate evaluations and well-written reports of findings. A formal review is quite different from an informal reviews, which involves the sharing of opinions between reviewers. The formal review is designed to provide reliable information about technical matters and may be more suitable to be considered as a testing technique.

If rigorously enforced and used as the configuration management status monitoring points, these reviews ensure that the data placed under project control meets project requirements, standards and conventions.

4.1.3 Unit Testing

Unit testing focuses on the smallest unit of a software system – the module. It tests whether the units function properly with respect to the detailed design description.

Unit testing basically consists of the tests for the *interface, local data structures, boundary conditions*, and *independent paths* within a module. The module interface is tested to show that data correctly flows into and out of the program unit under test. The local data structure is examined to assure that data stored temporarily maintain their integrity during all steps in an algorithm's execution. Boundary conditions are tested to assure that the module operates properly at boundaries established to limit or restrict processing. All independent paths(basic paths) through the control structure are exercised to show that all statements in a module have been executed at least once.

Unit test planning occurs in parallel with the derivation of detailed design, and the unit test plan should be documented into a *unit development plan.* Unit testing is normally conducted in the implementation step. After source code has been developed, reviewed and verified for correct syntax, unit test case design begins. The *driver* and *stub* modules are usually needed to aid the unit test. A driver module is used to simulate the module which call the module being tested and a stub module is used to simulate the module which is called by the module being tested.

The *Unit Test Walkthrough* is the last informal review of the unit testing conducted by the developing organisation [92]. This will mainly check:

1. The code meets the functional, performance, interface, and design requirements.

2. The unit test plan requirements have been completed, the test execution has been in accordance with the requirements of the plan.

After completion of the unit test walkthrough, the unit design, code and test information as defined in the unit development plan is baselined and is placed under configuration management and control.

4.1.4 Integration Testing

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested modules and build a program structure that has been specified by design.

At the integration testing level, modules are integrated in a hierarchical fashion tracking the execution sequence of the software within a subsystem. Qualified units are integrated into an operational configuration, and data relationships and internal execution characteristics, including performance, are verified against the subsystem design. Integration testing is conducted using versions of software formally controlled by the project through configuration management [32].

The integration test plan should be established at the stage of subsystem design and incorporated into the documentation of the *Subsystem Development Plan.* Integration testing is normally performed by an *incremental integration* approach which involves two main strategies: *top-down integration* and *bottom-up integration.* These two strategies were described in section 3.2.

All problems uncovered during integration testing should be documented and corrections should be formally controlled and tracked. The released software should be saved in a project library. Completion of final functional integration is a full qualification of the

software subsystem.

4.1.5 System Testing

System testing begins when integration testing has been completed. It is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. In [50] system tests are categorized as *requirement-based testing*, *performance testing* and *design-based testing*.

Requirement-based system testing concentrates on demonstrating system functional capabilities. Performance testing is the test of system performance capability. One of the performance tests has been given the special name of *volume testing*. A volume test is planned with the specific objective of determining how many transactions or how many records can be operationally supported. Other special performance tests include tests of system documentation (usability, completeness); system reliability(failure analysis, stress testing, operational testing); and system maintainability(documentation, time to make standard changes). Design-based system testing is used to test software design. Most of the tests derived from the design are suitable as system level tests and should be included in the system test set.

The planning for system testing starts with requirement analysis and may be continued in the next phase of system design. The planned tests should be documented in the *system development plan*. During system testing, corrections are made by the software organisation and regression tested and approved prior to integration into the test configuration. System testing ensures that a qualified software system is in operation.

4.1.6 Retesting

The need for the continuous modifications of software after initial delivery is one of the reasons for software maintenance. It has been stated [section 3.6] that about 60% of the total software life cycle costs is spent on maintenance. Based on this figure, software maintenance deserves individual emphasis, however it has historically not been given appropriate attention.

Changes required during maintenance may be due to [57]:

- Errors not discovered during original testing;
- Failures outside the data processing system, which can be corrected by software changes;
- Changing requirements for use of system;
- Modification to make people in system more efficient;
- Modification to make hardware and software more efficient; or
- Modification to algorithms to take account of experience with real data.

When a change is attempted to an item of a software system, three steps are usually involved:

1. Design the changes. This requires understanding the system or program logic; understanding the intended new result.
2. Build the changes. This involves actually changing the code and modifying the system to conform to the new design.

3. Implement changes and install a new system version. This involves replacing the changed items in the project library, ensuring that users are familiar with any new impacts and updating documentation.

When software is modified, a retest must be performed to ensure the modified software is correct with respect to the requirements and there have been no unintended changes made to the original system function. These retest activities are in general observed in three categories [38]:

- *Confidence Testing.* In this activity the main intended effect of the change is demonstrated.
- *Localised Testing.* This concerns that a check is made to see that all expected effects of the change are indeed observed. If the change is sufficiently small, it may well be possible to ensure during this testing that all modified code is executed. This is rarely possible for a system as a whole.
- *Non-Localised Testing.* This is to check the operation of the system as a whole and reveal any unintended side-effects of the changes.

Regression testing, embracing these three aspects, is a technique used in software maintenance to deal with the retest problem after software modification, which is described in sections 3.6 & 3.9.3.

4.2 Test Data

In this dissertation, the test data is defined to be all the data produced or used throughout the testing process, such as a *test plan*, a *test design specification*, *a test case*, *a program specification* and *a program*. This section focuses on the discussion of the characteristics of a test case, a program specification and a program.

4.2.1 Test Cases

A test case, consisting of a description of the input data to the program and a description of the expected output of the program for that input data, is a key element in software testing [81]. The purpose of a test case is to execute a program under certain conditions. Generally, all test cases belonging to a project are categorized in various levels according to testing levels(e.g. unit testing, integration testing and system testing) and defined in the different levels of test plans.

Test cases can be considered as software elements with the same characteristics as general software. The life of a test case also needs to pass the phases of analysis, design, implementation, execution and maintenance, which is in accordance with a *testing life cycle* described in section 3.2.

On the other hand, test cases have their own characteristics which make them different from other software elements. It can be easily recognised that the quality assessment of test cases is as important as program testing. However, a reliable test case has been defined differently from a reliable program [22] although both of them are categorized as software. For the reason that test cases can be derived either from specifications or from programs, test cases are usually required to be changed whenever a program or a specification changes.

The test case used during the maintenance phase evolve from the test cases used during development. However, new test cases created during maintenance must be included in the test suite to reflect changes to the system's specification, and redundant or irrelevant test cases must be eliminated. Recently, several techniques have been proposed to cope with the problem of how to identify obsolete test cases after a software modification [71].

4.2.2 Program Specifications

A program specification is a description of the desired program behaviour. Initially, a detailed functional specification of what the system should do can be developed from the requirements. The specification can then be verified against the requirements early in the development of the software.

Currently, program specifications are most commonly written in a natural language. However, using a natural language leads to specifications that are vague and ambiguous. Although such specifications do aid in detecting errors early in the development process, the imprecision of an informal specification leads to misunderstandings, both in testing the specification against the user requirements and testing in the implementation against the specification. Therefore, many people have argued that a more formal approach to specification is required.

There are a number of techniques that have been developed to aid in the writing of precise specifications. One technique is to use mathematical notation to document equations or algorithms whenever appropriate. Another technique is to construct tables of all the input and output variables and group them into some logical fashion [102]. Recently, more efforts have been made to produce a formal language which could be used to transform formal requirements into precise specifications. This precise specification can also be used to verify the resulting programs. Several well-known specification systems have been described such as *ISDOS, CLEAR, OBJ, GYPSY*, and *AFFIRM* [41]. VDM [59] and Z [106] are formal specification languages provided to support systematic software development and software verification.

Program specifications associated with a particular project may change during the project life cycle. These changes may take place for various reasons. One obvious case is when users add new requirements to the system. Regression testing is attempting to solve the problem of how to select test cases and how to keep test plans up to date after the

program specification has been changed.

4.2.3 Programs

A program is an implementation of a (program) specification. It is a vehicle for communication between humans and computers. Many programming languages have been developed for writing programs. Generally, all programming languages can be characterized with respect to three topics: *data typing*, *subprogram mechanisms*, and *control structure* [92].

Programming language characteristics and coding style can profoundly affect software quality, testability and maintainability. The effect of programming language characteristics on software testing and maintenance is a large and difficult subject of research. There is no question, however, that technical characteristics that enhance code readability and reduce complexity are important for effective testing and maintenance.

Automatic programming systems have been provided for transforming a precise specification into a program written in a specific language.

4.2.4 Relationships Between Test Cases, Specifications and Programs

Based on the characteristics of test cases, specifications and programs described above, the relationships among these components can be summarised below:

- A specification is a basis on which both programs and test cases can be generated.
- When a specification has been changed, programs and some test cases also need to be changed.
- The test cases can be derived from programs.

- When a program is modified, the test cases based on the program should be reselected.

4.3 The Need for Software Testing Management

A frequent criticism of current testing techniques is that they are less effective and efficient for large scale software. One of the main reasons for this is that although the techniques developed for a certain testing strategy are said to be powerful, the testing activities in each phase of the software development cycle are poorly managed. Generally, software testing management can be divided into three aspects: *the management of the testing process, the management of testing organisation and resources*, and the *management of test data.*

There have been a few methods proposed to manage the testing process, testing team and testing resources in the literatures. Miller [78] suggests that:

- The management monitoring of the testing process should indicate clearly at the outset the importance of a wide spectrum of information that relates to the testing process. The simplest method is to ask testers to share access to daily test progress reports.
- Testing is a labor-intensive activity, and choice of a testing team can be of crucial importance.
- The management can monitor testing progress indirectly by keeping track of the computer resource devoted to testing activities.
- Coverage measures and complexity measures are effective techniques for management, particularly for assessing the level of testing that will be required for determining the budget allocations needed to effectively complete a testing activity that

is mid-way to completion.

- Management can also be sensitive to the psychological interplay between program testers and program developers.

Evans [32] addressed the approaches to a productive software test management. He described how to plan, manage, and control the integration and testing of a multi-subsystem system configuration. The careful planning for the testing process was mainly discussed as a vital factor for testing success.

The research described in this dissertation, however, is mainly concerned with the management of test data.

In the following sections, three problems associated with testing will be discussed. These problems serve to show the need for the management of test data.

4.3.1 Difficulties in Early Planning for Testing

Early planning testing is a difficult activity. It requires that the project managers have an early understanding of what is to be accomplished. Without this understanding, it will be difficult to project how the testing process is to be managed and controlled; to estimate resources needed; to plan for applying the resources and doing the work; or to develop a realistic test schedule.

Even initial project planning is often an imprecise process based on incomplete information, poorly specified and misunderstood requirements. As a result, what normally happens is that the allocation and commitment of resources to the various test levels are confused and there is little way to ensure a smooth flow of data, effectiveness of test data integrity, and a clear transition of responsibility.

4.3.2 Large Amount of Data

The testing process is the constant checking of one developed item against another(e.g. to review the design specification against the requirements). It involves enormous amounts of data, and this data is in a state of constant change so that a number of versions associated with every data item are produced. Therefore, test managers and testers usually need to determine the state of the testing process and make a comparison of the data produced. This can be controlled only if the status of each data element is known at all times.

In comparison with controlling the large amount of data versions produced, the relationships among these data items are more complicated to maintain. Usually, these relationships are loosely coupled. For instance, when a function needs to be changed in the specification or new features are added into a specification, a caucus must be held to determine what activities must be reinitiated, what program modules and manual pages will be affected, and what test cases must be reselected. This management level information is more likely to be held on paper or in people's minds and be exchanged orally without records. For a project of long duration, staff turnover is normal and effective project management becomes threatened due to the lack of sufficient information.

4.3.3 Testing Software Changes

When a software item is changed, testing must be performed to make sure that the changed software functions correctly with respect to the specification. One of the major problems in testing changes is test selection which is concerned with how to select which test cases to rerun after a modification. It is important that these test cases be selected systematically, because executing an entire test suite to validate a few modifications can consume large amounts of time and computational resources and involve many people, and it is unreliable to exercise a system by selecting test cases intuitively or randomly. When the test cases have been determined to be used for retesting, the old test plan must be updated so that

it can be used for the next cycle of changes and regression testing.

Under the present state of maintenance testing technology, effective regression testing is seldom possible or complete. The problems are that the input test cases are usually stored and organised in a variety of different file and data formats, and storage media. Furthermore, complicated procedures are needed to enter the initially large set of baseline test cases and verify the individual responses from the software under test. Generally, the verification team has no way of correlating any of the functional requirements to the associated test cases, leading to a situation in which the testers do not know if the results obtained by executing the test descriptions are correct and coincide with the required user specification [47].

It is believed that the support of software configuration management will greatly benefit the solution of the above problems. In practice, it is difficult to construct a cost-effective and usable software system without a good system of configuration management in place. The next section describes the aspects of software configuration management which are applicable to the software testing process.

4.4 Software Configuration Management in Context of Software Testing

As discussed in chapter two, software configuration management is the complete mechanism for controlling and recording the status of all deliverables, their relationships and their changes. All software items concerned with testing should be subject to configuration management control.

Software configuration management is a large subject with a literature of its own, but there are three aspects that concerns testing: *change control, version control* and *record keeping*, which are discussed in the following subsections:

4.4.1 Change Control

Software development is a process of change. In the area of testing, there are four areas in which change control is required:

- the need to keep the *Test Plan* up to date when designs and other deliverables change.
- the need to modify code, specifications, test cases and other documents when errors are revealed by testing.
- the need to retest items whose specification have been changed.
- the need to identify the effect of change when the data has been modified.

The change control process should take place when there is a need for modification after testing has been conducted. A simple example could be seen when some errors are found in dynamic testing, either the item being tested or the specification on which the test case is based or both will require modification. The change control process is especially useful for the revalidation process in software maintenance. The revalidation process is concerned with the test of modified software, and ususlly involves the problems of reselecting the test cases and updating the previous test plan.

4.4.2 Version Control

Most deliverables will evolve iteratively and thus go through several versions during system development and subsequent maintenance. This may occur simultaneously for different deliverables in the life cycle. Testing cannot be done sensibly unless the version of each item relevant to the test is known and confirmed as the correct version for the test.

Version control is especially important when a subsystem or entire system is being built and then tested. Each version of the built software contains a particular version of each of its components, and the management of this process requires thorough record-keeping of these versions, which is described in the next section.

4.4.3 Record-Keeping and Traceability

Neither change nor control can function without adequate record-keeping. It is suggested in [86] that the best criterion for record-keeping is that of traceability, the ability to establish an audit trail of relevant information. In the context of testing, this can be interpreted as the ability to trace an error to its source. To provide for traceability, each item in the chain should provide full cross-referencing of its components to the items on which it is based. For example, each component in the *design specification* should be cross-referenced as far as possible to those features in the *system specification* which it implements.

Another aspect of traceability in testing is the ability to follow the progress of errors revealed by testing. For each error found, it should be possible to trace the process of correction through the audit trail from the point where the error was found to the point where it was corrected.

The third aspect of traceability in testing deals with the links of test cases with the specifications and programs. For each test case designed, it should be possible to trace a specific test case to the system specification on which the test cases are based.

A final criterion of traceability is the ability to trace the static and dynamic tests that were performed on an item after each change in its development history.

4.5 Summary

4.5.1 The Purpose of This Chapter

With the aim of applying SCM techniques to the testing process, an investigation into various aspects associated with software testing management has been conducted. This chapter is devoted to presenting such an investigation. The objectives of this chapter are concluded here:

- To address the testing activities at different levels across the software life cycle, together with the management aspects which should be applied to each testing activity.
- To stress the importance of early test planning.
- To define the test data to be applied in the SEMST system and specify the relationships between the data.
- To analyse the need for software testing management.
- To discuss SCM techniques concerned with the software testing activities.

4.5.2 Combining the Testing Process with SCM – a Refinement of Previous Discussions

The previous chapters of this dissertation have presented:

- the objectives of this research project;
- software configuration management and testing as two disciplines of software engineering;

- the tools and techniques of software configuration management and testing.

Based on the definition of software testing described in section 1.2, this dissertation regards testing as a broad and continuous activity over the software life cycle, ranging from informal design reviews through rigorous test analysis to formal proofs of correctness.

Once the basic features of SCM and testing discipline are understood, the research and development activities will then concentrate on the methods for applying SCM techniques to the testing process. The significance of such methods has been analysed from the following aspects:

- Early test planning is important but seldom complete in the actual software development [section 4.1.1 & 4.3.2].
- Review activities can ensure that a testing process is under SCM methods control [section 4.1.2, 4.1.3, 4.1.4 & 4.1.5].
- The large amount of test data and their versions produced during the testing process must be managed and controlled [section 4.3.2].
- The retesting process requires that the previous test cases and their execution results be stored and maintained [section 3.9.3 & section 4.3.3].
- Software testing is a process of change. Change control must be applied to software testing [section 4.4.1].
- To provide the long-term maintainability of software systems, record-keeping and traceability are required for software testing [section 4.4.3].
- Measurements of test coverage and complexity are considered as effective techniques for software testing management[section 4.3] [78].

4.5.3 Limitations of Testing Management

The remaining problems with the current SCM techniques were described in section 2.5.1. These problems may affect the application of SCM to testing. For example, SCM techniques may be difficult to use in conjunction with distributed and heterogeneous software systems as such systems make SCM a more difficult task.

On the other hand, management of software development is usually regarded as a support technique used to facilitate the development of a complex software project on schedule and within budget. However, the success of a software system development is not dependent solely on the management mechanisms. It has been stated [20] that "management review of development progress will not ensure successful completion". The advanced tools and techniques used to aid the system development are very necessary, and these should come first. In the area of software testing, management support will not necessarily result in a successful testing process unless there are good testing techniques or tools which are also used in the testing process.

Chapter 5

Survey of Previous Work and Analysis of SEMST Requirements

Introduction

This chapter presents a survey of previous work associated with the management of the software development and testing process, and analyses the requirements for SEMST. Much effort has recently been devoted to the development of integrated database support techniques and project management tools. These tools provide systematic approaches to managing a software development process. However, little attention has been given to provide the tools and environments to support the management of software testing. By evaluating previous work, the benefit of developing SEMST will become clear.

5.1 Integrated Software Engineering Environments

In the last few years, much emphasis has been given to the research and implementation of *Software Engineering Environments*(SEEs). The purpose of a SEE is to support users in their software development and maintenance activities. These environments range from simple tool kits to fully integrated tools supporting a software engineering method. It can be concluded that SEEs can offer significant productivity improvement, higher software quality, and better project management and control.

In general terms, SEEs have been viewed as being composed of two distinct classes: *program environments* and *project support environments.*

Programming environments concentrate on the support of the coding stage of the software development cycle. Some examples of this kind of environment are: *APSE* – Ada Programming Support Environment [77], the *Interlisp* programming environment [111], the *Cedar* environment [110], and the *Smalltalk* environment [42]. These systems incorporate tools for editing, parsing, debugging, and documentation.

The project support environment, recently referred to as an integrated project support environment(IPSE), is an environment to support the whole range of development activities carried out in a project, including programming-in-the-large tasks such as configuration mangement and programming-in-the-many tasks such as project and team management. This means that the integrated environments should provide a homogeneous support for specification, design, development, testing, management of versions and releases, distribution activities, configuration and customisation, error reporting and measurement collection. There are quite a few of this kind of environments developed in recent years. Several well-known examples are: *Gandalf* – a integrated software engineering environment [45], *PCTE* – a Portable Common Tools Environment [13], *ECLIPSE* – An Integrated Project Support Environment [9], *ISTAR* – second generation of IPSE [27], and *Arcadia* – an advanced software engineering environment [109].

5.2 Management Systems

The management system, classified as *project management, process management* and *object mangement*, is an important part of the integrated project environments described above.

Project management involves cost estimation, resource estimation, and scheduling. Traditional project management systems were developed by using a Gantt chart, CPM or Pert algorithm. Recently a DesignNet model [73] has been produced on the basis of a Petri net notation to support rescheduling and reinitiation of the project management.

Process management is concerned with the industrial approach to the software production. In the context of a software factory, the integration of tools differs from the traditional approaches in that it includes the integration of people and their "corporate knowledge": their organisation, their rules and polices and their methods. The ESF [35] is an example of this type of system.

An object management system, sometimes referred to as a data management or information management system, provides the support for managing different kinds of data ranging from source code, executable code to documentation, test plan and test data. Such a system has been identified as the core of any automated SEE and is vital to the success of a Computer-Aided Software Engineering(CASE) tool.

Traditional SEEs are built on the basis of a file system or a database system. There has been criticism that such file and database systems are inadequate for handling the large amount, wide kinds of types and complex relationships of data in the real world. Recently, an object management system has been proposed to overcome the weakness of previous work on data management, and it has been drawing a lot of research attention.

The next section describes the characteristics of an object management system.

5.2.1 Object Management Systems

A large software product consists of a wide variety of objects. It consists not only of source, object and executable code objects, but also of requirement, specification, design, schedule, test plan, test data, and other documentation objects. The systems to manage these objects must address a number of problems. These problems include [5]:

- Storing multiple versions of data objects(e.g. multiple releases of software and documents);
- Storing large, variable-length objects whose internal structure is hidden from the object management system(e.g. programs and documents as text);
- Creating multiple objects representations to allow different languages, tools or hardware;
- Producing flexible and powerful operators, such as operators on directed graphs to manipulate syntax trees, flow graphs, and dependency graphs, including set-at-a-time capability;
- Providing flexible data types to store arbitrary types supported by the programming languages.

Most environments have been built on the basis of a traditional file or database system for managing the objects associated with a project. In a traditional database, the information is usually modelled as records. Relationships among data entities are constructed through primitive reference to related records [73]. However, in practice, even the most powerful database systems are inadequate for the data handling requirements of CASE [5], which are described above. It is now believed that an object management system can deal with the above problems and enhance the environment support for change, integration, software reuse, and cooperative work by multiple people. The advantages of an object

management system can be seen in its object-oriented approach to capturing relationships among objects. It allows more complicated relationships. For example, an association between entities may itself be considered as an entity and further relationships can be built upon this entity.

According to [109], an object management system for a SEE should provide support for: *types, relationships, persistence* and *concurrency and distribution*, which are described in the following section.

Type Systems

A type system is viewed as the primary mechanism for describing and maintaining objects. An object management system should be able to enforce the type system, hiding the internal structure of typed objects behind well-defined interfaces and strictly controlling the operations that can be performed on those objects. If all objects are instances of abstract data types, it is easier to share objects or to change their implementations. Thus, basing the object management system on a typing system that fully supports data abstraction will result in environment flexibility and software reuse.

Typing of objects in programming languages is a well researched area and generally considered to be of significant benefit to software engineering. However, current approaches to object management in SEEs are far from providing full support of typed objects. Typically, the components of a product are treated simply as files and tools are viewed as operators applicable to the contents of those files. Usually in such systems, only a predetermined and limited number of different kinds of components and operations are available [109]. *Make* [34] and *Odin*[64] use file names extensions as a weak form of typing mechanism. It also allows users to define which tools could operate on or produce files of various types. The *System Modeller*, developed as part of the *Cedar* system [66] used the term "object" for referring to the files containing product components but did not treat the

objects as instances of abstract data types. *The Common APSE Interface Set* (CAIS) defines a system model with three kinds of nodes – file, structural, and process, but does not treat those nodes as typed objects. *Gandlf*'s SVCE mechanism employs strong type checking to determine consistency of syntactic units during version control. Recent work on rich type systems, particularly in the system context of object-oriented languages, is also encouraging, but also still not mature.

Relationship Systems

Closely related to the ability to precisely define and maintain the typed objects in the environment is the ability to capture and maintain the relationships among those objects. Examples of relationships include those connecting various versions of a module, or those between the modules constituting a configuration, or those between a module and all the others that it calls, or those joining activities in a work breakdown structure [109]. Examples of tools that reason about or exploit relationships among objects include a version control system [100] [112], automated system building tools [34] and call graph analyser. Associated with the relationship system is a set of capabilities, such as consistency checking, derivation tracking, and inferencing.

Clearly indicating the relationships among an environment's tools and information structures could provide an easy way to modify the environment since the effect of changes can be determined. Furthermore, capabilities that rely on relationships, such as inference and derivation, can enhance environment integration by providing abstract type mechanism and allowing users to interact with the environment at a high level. Generic relationship capabilities can also enhance integration by providing a uniform set of capabilities across different kinds of relationships.

The previous work on building relationships in the environments is weak. There was little systematic method provided to manage numerous and complex relationships between

objects.

5.2.2 Persistence

Persistence support in an object management system means that it should be able to allow the objects to continue to exist beyond the lifetime of any of the tools or process programs that manipulate them and preserve the integrity of their types and relationships to other objects [109].

Current approaches to persistence are based on files or databases. Using a file system, a tool should be provided for converting the internal form of an object to an acceptable (e.g linear) external form and, when needed converting it back. Using a database system, the tool must make calls on the databases to explicitly store and retrieve information. These traditional approaches have the limitation that they are only applicable to a limited number of object types. Thus providing persistence for arbitrarily complex, typed objects is an important research subject.

5.2.3 Concurrency and Distribution

An object management system should be able to allow multiple users to work on the same software development project. This requires the support of concurrent and distributed capabilities. In a network of workstations, different members of a development project may simultaneously invoke the same or different tools to operate on one or more of the same objects. Thus, the object management system should have the ability to mediate concurrent use of objects and to maintain consistency of both the objects and their relationships.

A number of approaches for handling distribution and concurrency have emerged from programming languages, and file system and database research [109]. However, few of

them have been universally accepted. Some of the difficulties of providing this capability are discussed in [109].

5.3 Hypertext Systems

Hypertext systems [8] provides information management, in which documentation is displayed as a network of nodes connected by links. Such nodes can contain text, graphics, audio, video or can link to other software or data. The distinguishing feature of hypertext systems is that they allow a non-linear organisation (i.e. it supports the links between parts of the documentation for purposes such as explanation and comments).

In the past years, a number of research projects have significantly advanced the technology of hypertext. Some examples are the following:

- *Brown University's Intermedia* was the direct descendant of an early hypertext project called FRESS [21] by Nelson and Dam at Brown University in early 1970s. The institute for Research in Information and Scholarship (IRIS) at Brown University has developed hypertext systems for a variety of courses, including for the course of English Literature [125].
- *Carnegie-Mellon's ZOG* was a research project on information management, conducted in most of the 1970's and early 1980, for use in USS Carl Vinson. USS Carl Vinson is the largest aircraft carrier in the world. A commercial product, KMS, was derived from this project and is marketed by Knowledge Systems Inc. to run on Sun and other workstations[1].
- *Xerox PARC's Notecards* is a system developed on Xerox Lisp machines at the Palo Alto Research Centre(PARC) of the Xerox Corporation [46]. NoteCards provides an environment in which the electronic equivalent of 3" plus 5" note cards can be created

to contain both texts and graphics, and hypertext links can be created between the cards.

- *University of Kent's Guide* was initially developed in 1982 running on Unix by Brown at the Computing Laboratory of the University of Kent at Canterbury [11]. Guide was further developed by Office Workstations Ltd. as a commercial product for the Apple Macintosh and the IBM PC.
- *Apple Computer Inc.'s HyperCard* was developed in 1987. It is one of the most widely available hypertext systems at the moment.

Because hypertext systems are useful in large scale information management, it has been suggested that the integrated software engineering environments include support for hypertext [40]. The combination of hypertext system and software engineering environment can advance an integrated software engineering environment in a way that relationships between the data information can be controlled. Therefore, hypertext techniques should be adopted in a testing environment to manage and control testing data documentation.

5.4 Software Maintenance Environments

In the context of building software tools applied in the life cycle, attention has traditionally been focused on the design and development of new software. The maintenance and enhancement of existing software has received relatively less attention. However, there is an increasing recognition that maintenance of software is very expensive, so that there are now a number of researchers working in this area. A number of tools and environment have been produced to support maintenance activities.

The maintenance techniques have been classically viewed from technical and management perspectives. From the technical perspective, the maintenance task is thought of

being composed of four main activities [16].

- The first activity is understanding the software which is to be changed. The maintenance environment will provide several tools to assist in this activity such as extensive cross referencing reports and query capability into the maintenance database.
- The second activity is to incorporate changes into the software. Therefore traceability from requirements through design into the code should be provided in the maintenance environment to control software change.
- The third activity is the accounting for possible ripple effects as a consequence of the changes introduced in the second activity. A ripple effect analyser should be included into the maintenance environment to guide in ascertaining the ramification of the changes throughout the program.
- The last activity is the testing of changes. This testing must be done in a cost-effective way, minimising the number of test cases which must be rerun. The maintenance environment should provide mechanism to support this activity.

Examples of this kind of maintenance environment have been described [39].

From the management point of view, maintenance activities can be divided into two classes: *product-related* and *process-related.*

The product-related maintenance management systems include version and revision systems (e.g. RCS and SCCS, see chapter 2), change coordination systems (e.g. Infuse [89]), reuse support systems (e.g. Draco [84]) and configuration management systems (e.g. Make, DSEE, see chapter 2).

Process-related maintenance management activities [88] include *personnel management, resource management, subprocess scheduling, walk-throughs, quality audits* and *Plan-*

ning. These are generally done manually, although some machine aids have been developed.

As described above, the activity of testing software changes is one of main concerns of software maintenance environments. In order to support this activity, a software maintenance environment should be able to manage the test data which ranges from the old data used during development to the new data produced during maintenance. However, there are few research literatures which have described this aspect.

5.5 Integrated Software Testing Environment – TEAM

In [18], a support environment for testing, evaluation and analysis (*TEAM*) is introduced. The TEAM project started in 1986 when the authors recognised that there was no single testing or analysis technique alone that can provide assurance of reliable software, but the careful integration of a number of diverse testing and analysis techniques can achieve software reliability needs. Thus the TEAM environment has been designed to support the integration of and experimentation with an ever growing number of testing and analysis tools, such as data flow analysers, symbolic evaluators and debuggers. TEAM provides the interpretation, data flow analysis, and reasoning facilities to aid the understanding of the execution results of the tools. It has been claimed that the TEAM can offer:

- integration of diverse testing and analysis tools;
- extensibility of that tool set so that new tools can be easily added;
- experimentation with different approaches to software reliability, and
- full software life cycle testing and analysis.

The current version of TEAM is an initial prototype which runs on the Sun, Dec/Ultrix and Dec/VMS. It is addressed by the authors as part of Arcadia Environment.

The aspects of management and control over the tests are not well-addressed in TEAM, although it is described in [18] that TEAM uses an object management system–*PGRAPHITE* [122] to deal with the objects.

5.6 Test Management Support Techniques

Test management techniques, as a class of supporting test tools, are not used directly for testing purposes; rather, they provide support to the testing environment by increasing test effectiveness and control. These are summarised below.

5.6.1 Test Execution Aids

These are techniques which are applicable to the test phase. Commonly used test aids include *test scenario files* and *test drivers.* Some of these tools, such as the *simulators*, can be essential to conducting certain levels of maintenance testing, even though they may have been developed to assist initial software development [87]-[101] [75]. *Automatic development* and *verification systems* are emerging, which help to enhance error detection, facilitate test case generation, and provide structured testing environment [60] [93].

5.6.2 Documentation Aids

Large amounts of documentation are prepared to support formal testing. Documentation aids help to reduce documentation cost and facilitate document preparation and maintenance. These aids include *test editors, documentation and report generators*, and

automated logic and flow chart generators.

5.6.3 Test Controls

Test controls aid in controlling test configurations, data, and test conduct. Often used test control techniques include *version control, automated program libraries, critical path scheduling techniques*, and *data and file management tools. Unit development folders* [32] provide historical logs of completed tests, and *success criteria* or *test completion criteria* help to define when to stop testing.

Automatic test drivers, such as the TPL/F system [26], developed in the late 1970's, are tools to simulate an environment for running module tests. Its advantages include the standardisation of test case descriptions and ease of regression testing. The main drawback is the difficulty in learning and writing a test language [22].

Comparators are data and file management tools used in comparing two versions of data to identify their differences. The data may be program code, output of an execution, or data files [26]. The comparators can be used in the validation process to help limit the scope of reverification of revised software. Some example systems are described in [22].

In [72], *Assay* is introduced as a tool, on Unix, to support regression testing. The main features of the system include configuration control of the tests, the ability to continue testing after a mismatch had occurred, and the filtering and substitution of selected character sequences. It is important that Assay provides test case management and execution facilities with an intention of support for efficient management and execution of available test data. Unfortunately, Assay does not include the facilities for managing data relationships.

Recently, there have been publications associating testing with *Hypertext* support [76] [47] [79]. By implementing a testing system with hypertext support, users would be able to

store, retrieve and execute not only the relevant test cases, but also maintain the necessary functional and design specifications, establishing links between the different contexts such as test documentation, test case coverage statistics, and the source/object code of the associated software under test. However, there are few hypertext tools provided with the entire capabilities described above and the idea of a testing system with hypertext support is therefore at the stage of concept and research.

5.7 SDDB – System Description Data Base

The SDDB [31] has been developed as a repository of the REDO project. REDO, standing for REengineering, Validation and DOcumentation of systems, is an Esprit II project concerning with the methodologies and tools to facilitate efficient and high quality software maintenance. There are several maintenance tools which have been developed in REDO. The SDDB is used in REDO as a central database which stores all data relevant to the maintained application(e.g. reverse engineering), shared by the REDO tools, together with the appropriate links and relationships between such data. According to [31], the main functionalities of SDDB are the following:

- it provides a central store of all information about the reverse-engineered application,
- it provides explicit representation of the application structure and logic(i.e. the structural representation of syntax and semantics of the source code),
- it enables the REDO tools to manipulate applications which are language and environment independent,
- it supports the integrity and consistency of data relevant to the reverse-engineered application,
- it allows the users to access and update the integrated toolset,

- it acts as a means of communication between tools, and
- it enables version control of information during the maintenance activities.

The need for using such a central, shared database in REDO can be justified as follows:

- Software maintenance involves a large amount of complex data and a persistent data store is required for any realistic maintenance application.
- The various tools used to support software maintenance manipulate interconnected and overlapping data types and often deal with the same data instances. Therefore, the data should be stored so that it is accessible to the different tools.
- Due to the complexity and long time scale of maintaining a large software system, it is necessary to enable and control complex patterns of read and write access by multiple users.
- Data integrity is important when many tools share the same data and the central, shared database can enforce the data integrity.
- There are many functions common to many different maintenance activities. It is sensible to put these common functions in a shared database.

The SDDB is based on an Object Management System(OMS) with the Entity-Relationship-Attribute data model[31]. It has chosen the Tool Builders's Kit(TBK) in ECLIPSE[9] as its platform, which is a software engineering environment extending PCTE[13] with additional database facilities. UNIFORM has been developed as an intermediate representation of the various REDO applications, which allows a reasonably direct translation of the source code to be restructured. The hypertext technique has been employed for managing the links between the entities stored in the SDDB.

The items which can be stored in the SDDB are all the information about an application which is used or generated by REDO's tools, including source code, historical

life-cycle documents, diagrams, ad-hoc notes and links created by the maintainers, metrics, application data models and formal specifications.

The SDDB has been intended primarily to support the reverse engineering activities in software maintenance, so it has provided little help for the testing process. As a software maintenance environment, however, the SDDB has not claimed its support for regression testing.

5.8 Analysis of the Requirements for SEMST

The purpose of system requirements is to state the functionalities of the system. Without system requirements, the development of a software system becomes chaotic. This section presents an analysis of the problems attempted to be solved by SEMST. It acts as the bridge between the related work, described in the previous sections of this chapter, and the requirements for SEMST, described in section 2.5.2. To avoid repeating what has already been described, this section focuses its attention on addressing the motivation for those requirements.

5.8.1 Motivation For SEMST

The SEMST system is intented to use software configuration management techniques to aid the testing process. The major functions that SEMST will supply include version control, relationship control and traceability of the test data. These requirements are motivated partly by the need for automated support for the testing process; partly by the demand for the long-term maintainability of software products; and partly by the need for a regression testing database. Each of these is elaborated in turn.

5.8.1.1 Controlling the Testing Process

SCM techniques manage the evolution of software systems by controlling and recording the status of all deliverables, their relationships and their changes in the software life cycle. It has been well-recognised that application of SCM methods is vital to the system development and maintenance. Much work in the past few years has led to considerable achievements in this field.

Unfortunately, there have been very few texts or research contributions which specifically address the methods of applying SCM to software testing, and very few such tools have been developed.

With the progress of software engineering research, the view of software development has been changed from focusing solely on implementation to a wider scope which includes analysis, design and other activities. Similarly, software testing has been gradually viewed as a broad activity which is performed at each stage of the software development life cycle rather than just a follow on activity after the coding. In fact, whenever a software system evolves, a relevant testing activity should be performed in order to ensure that the software system evolves correctly with respect to its requirements. Therefore, it can be easily understood that the evolution of the testing process follows the evolution of a software system. Section 4.4 has addressed the SCM areas that should be applied to the testing process.

The problem is that the testing process becomes difficult to carry out when coping with large and complex software. To determine the test cases for testing software changes, the tester normally needs to understand the relationships between the current versions of system components. However, in large and complex system, developed over a long period, such information about the system is usually not readily available.

Therefore, SEMST is attempted to help this problem. It is required to maintain all

versions of the specifications, test cases and programs produced during the project life cycle, and to control the links between these data items. Thus, the users can obtain the traceability between these data items over the whole testing process.

5.8.1.2 Supporting the Long-Term Maintainability of the Software System

Associated with the evolution of software systems is their long-term maintainability problem. What normally happens in a real software development is that the documentation is not maintained, and the traceability of the code to the system design is lost. This results in a situation that the software system becomes difficult to maintain after the system has evolved over a long time.

By emphasising the management of test information, SEMST would enable the SCM techniques to be embedded within the software development and maintenance. It could help to ensure the system's documentation and design information are maintained along with the code. Furthermore, the versions of these components and their relationships would be controlled by SEMST. This will benefit a testing process, particularly the revalidation precess during software maintenance. Other maintenance activities may also be helped by SEMST.

5.8.1.3 Regression Testing Database

The design of SEMST is strongly influenced by a repository required for regression testing. The problems associated with regression testing have been previously described. In order to solve these problems, a database which supports automatic regression testing is needed.

Basically, a regression testing database should store three test components namely specifications, test suits, and programs. The important requirements for the regression testing database are that the history record of these components must be maintained, and

all possible links between these three components must be identified and controlled.

So far, however, no regression testing database which satisfies the above requirements has been found. SEMST can be essentially considered as a database for supporting regression testing.

5.8.2 Review of the Previous Work

To analyse the requirements for SEMST, the related work should be evaluated. This section explains why SEMST is needed by reviewing the previous work which has been described in the previous sections.

1. Integrated Software Engineering Environments[section 5.1]
   An integrated software engineering environment is designed to support various software development and maintenance activities. It consists of the mechanisms for software configuration management and software testing. However, management information for software testing is not fully captured in the current integrated software engineering environments, as the requirements made for these environments are too broad to focus on testing management.

2. Object Management Systems[section 5.2]
   Most modern software engineering environments are built on the basis of an object management system(OMS) because an OMS has the advantages in supporting change; integration and reuse of the software systems; as well as supporting cooperative work by multiple people. Based on an object-oriented approach, the OMSs enable the control of the typed objects with inheritance; complicated relationships among the objects; data persistence; and the concurrency and distribution. From the above, the OMSs can be used to provide the management and control of test data used in the testing process. However, the current OMSs provide little systematic method to support rich type system and complex relationships between the

objects. To provide the ability of persistence, concurrency and distribution in the OMSs, much research work is still needed. The few OMSs to have been developed give no emphasis to supporting the management of testing.

3. Hypertext Systems[section 5.3]

   Hypertext techniques have been recently developed to control the relationships between the life-cycle documentation, based on the notion of links and nodes. Obviously, hypertext techniques can be used to manage the test documentation and its links. However, the current hypertext techniques are not concerned with change effect on the data and the links. The links which have been created between data would become insecure after a change to the data. But the current hypertext techniques do not provide an approach to deal with this problem. There have been few hypertext systems practically available. Moreover, version control is not actually well-provided in the current hypertext tools.

4. Software Maintenance Environments[section 5.4]

   One of the activities that the software maintenance environments should support is testing changes. To aid this activity, a maintenance environment should be able to manage the old test data generated or used in the system development as well as the new test data generated or used in the system maintenance. The history of the system documents and their relationships should also be managed in a maintenance environment. Unfortunately, few research issues have been found to address the above in the software maintenance environments, and currently, the available software maintenance environments are few.

5. Integrated Software Testing Environments[section 5.5]

   It has been recognised that no single testing technique alone can assure a reliable software system. The integrated testing environments have been proposed to integrate a number of diverse testing techniques in order to achieve the software reliability requirements. TEAM[18] is one of the examples of the integrated testing environments. There are however, some limitations to TEAM. For instance, although it has

been claimed that TEAM can support full life cycle testing and analysis, the mechanism for regression testing is not actually provided in TEAM. In addition, TEAM does not include the ability to control test data versions and their relationships.

6. Test Management Support Techniques[section 5.6]

   Test management support tools provide their support for the testing activities in many ways. Some tools perform the function of test execution coordination. Some tools provide a controlled environment in which testing can take place. There are also a number of such tools which aid the test documentation. *Assay* [72] is a support tool for regression testing, which is aimed at providing efficient test data management and execution facilities. It has been developed to use SCM techniques to control tests(e.g. it provides test cases version control). However, Assay does not provide the management of links between test data. Unlike these previously developed test support systems, SEMST emphasizes its support for the whole testing process and its evolution. It is designed to manage the test data associated with each testing activity across the software life cycle.

7. SDDB[section 5.7]

   The SDDB, developed in REDO, is a central, shared database in which all the data relevant to maintenance and their relationships are stored and controlled. The SDDB belongs to the class of software maintenance environments which have been designed to provide sophisticated mechanisms for supporting the maintenance activities. In comparison with the SDDB, SEMST is proposed as a system developed within REDO, which focuses on control and management of the testing process. Despite that the SDDB has been described to provide the support for various maintenance activities, it is actually focused on the support for reverse engineering. Regression testing is a maintenance testing technique used to ensure a reliable maintenance activity. However, the SDDB has not addressed problem of supporting regression testing. Therefore, SEMST is to be developed as a testing support environment which may augment regression testing systems in REDO, and thus it may be as a useful part of the REDO environment.

5.8.3 Design Criteria for Prototype SEMST

In association with the requirements for SEMST, this section identifies the design criteria for the SEMST prototype version.

- SEMST should be built as a database system which includes basic SCM abilities.
- SEMST should be able to load three types of test data, namely specifications, programs and test cases into its database.
- SEMST should manage the specifications written in a formal or informal language.
- SEMST should manage the test cases derived or used over the entire software life cycle.
- SEMST should maintain all versions of these three components.
- SEMST should enable the user to retrieve and update these components in the database.
- SEMST should allow a set of test data to be baselined(i.e. a release).
- SEMST should be able to identify and control the links of test cases with specifications and programs.
- SEMST should report the changes made to these test data.
- SEMST should keep track of the state of links between these data and report insecure links caused by the changes to these components.
- SEMST should provide the user with information about the affected test cases resulting from the specification and program changes.
- SEMST should allow multiple users.
- SEMST should provide data security control, preventing multiple people from updating the same data item simultaneously.

- SEMST should provide a user-friendly interface.

5.9 Summary

In this chapter the tools developed in the area of software engineering have been discussed from the point view of data control, management and maintenance. Although many current software environments claim applicability over the entire software life cycle, their effectiveness during the testing process can be greatly improved. Actually, the management and control information provided for the software testing process is not fully captured in these environments. Therefore, it is necessary to develop a management environment to support the testing process. This support environment can then be built within a software engineering environment. The analysis of SEMST requirements has indicated the significance of developing SEMST.

Chapter 6

SEMST – A Support Environment for the Management of Software Testing

Introduction

The SEMST system is aimed at managing the testing process with SCM support. Generally, testing management is a broad term which can include management of the testing process, management of test data and management of the testing organisation and resources. The current SEMST focuses on managing and controlling the test data produced in the project life cycle. The test data involved in SEMST is the following: specifications, test cases and programs. In this chapter, the SEMST system is presented by looking at its properties and design. An example of applying SEMST is also described.

6.1 SEMST Capabilities

SEMST supports the management and control of test data (e.g. specifications, test cases and programs) produced in a project development cycle. Its support covers all levels of software testing, including unit testing, integration testing, system testing as well as regression testing.

It possesses the ability to store, retrieve and update all the versions of test cases, specifications and programs associated with a project. The links between these objects can be established and controlled. When a modification has been undertaken to one of these items, the system will provide users with change information so that the items linking with this changed item should be given attention. In this situation, the current state of the links between them is defined to be *insecure*. The system allows multiple users to access the database and some security checking will take place in order to prevent two people from updating the same file at the same time.

The overall capabilities of SEMST can be categorised as: loading the data into the system, maintaining the versions, retrieving and updating the data, managing the links between the data, and controlling security over the data. The following sections consider the properties of SEMST in more detail.

6.1.1 Loading Data

There are three categories of data stored in the SEMST database, namely specifications, test cases and programs. The specifications supported by the SEMST prototype are those with a style which is rule-based or functionality-based, although specifications written in other styles are also acceptable to the system. The test cases are usually described in a variety of formats. In SEMST, the test cases are represented by the description of the input data, the description of the output data and other relevant attributes. In SEMST,

a program is stored together with its attributes. The program's attributes are generated from a static program analyser, which are usually represented by a number of tables(e.g. subroutine table, branch table and path table). SEMST provides a connection to a static program analyser so that the program attribute tables can be loaded into the system. The data representation in SEMST is presented in section 6.2.3.2.

Two ways have been provided to input the data into the SEMST database. If a data file already exists in the machine, SEMST is able to convert this file into the database, when given the complete path name of the file. SEMST also provides a facility to guide the users to input new data into the system. This means the users can input the data by following the instructions provided by the system.

6.1.2 Maintaining Versions

This is based on RCS – A Revision Control System [113] [section 2.4] on Unix. SEMST provides basic version control, history management and configuration management mechanisms to maintain all the data stored in the system. SEMST keeps track of any changes made to a file and controls all versions of the files. Any version of a file can be retrieved from the system database provided that the version number is given. The latest version can be retrieved by using the default option.

The first version of a file is numbered 1.1 and successive revisions are numbered 1.2, 1.3, etc. The first field of a revision number is considered as the *release number* and the second one the *level number*. A release is a software deliverable or an end product which may be baselined [see section 2.2.2]. SEMST supports releases, so for example the user can define all the latest versions of test cases used for unit testing to be a release, and users are allowed to retrieve any version of a file from a release. SEMST also allows the retrieval of a branch version of a file by giving the branch number. Suppose in SEMST a file version sequence is as follows:

1.1—1.2—1.3—2.1—2.2—...

A branch occurs when a modification has been undertaken to any version before the latest version. The first branch starting at 1.3, for example, has a number 1.3.1, and the revisions on that branch are numbered 1.3.1.1, 1.3.1.2, etc.

6.1.3 Retrieving and Updating

SEMST supports the retrieval of any version of the data from the system database. The users are allowed to retrieve a test data item by providing the system with the file name/item's identifier and the version number. The system uses the file name/item's identifier as a keyword to make a search among the data in the database. If the search is successful, the system will then check the version number. Thus, the users can obtain the content of the data item retrieved on the screen as long as the file version related to the version number exists.

The users can modify the data retrieved from the system. When a change has been undertaken to a file, the new version of this file must be brought into the system and the system is supposed to conduct the tasks relevant to the change, which will be described in the section 6.1.4.4.

6.1.4 Managing Links

In order to allow the traceability between the test data, a link management mechanism has been provided in the SEMST system.

6.1.4.1 The Link Concept in SEMST

Links are used to represent the relationships between several objects. The links among objects can be defined in many ways according to their particular purpose. For instance, links can be established between data files to indicate the versions. In SEMST, the relationships between test cases, specifications and programs are maintained by creating links among them. The links are managed on the basis of the *identifiers* associated with these components. SEMST requires that each data element be given a name as its identifier. In SEMST, the relationships between test cases and specifications are represented by links of the test cases with the parts of the specification[1], and the relationships between test cases and programs are represented by links of test cases with the program's attributes(e.g. the procedures/functions, the paths, the branchs, and statements etc.). A link can be given between two elements when either one element is a derivation from another or one element's execution can cause another to be activated. For example, if a test case is generated by means of a specification-based testing strategy, there must be a link between this test case and a part of the specification. When using a test case to run the program, if the execution fires a part of the specification, we can say that there are certain links between this test case and the specification. A part of the specification may have links with several test cases and a test case may have links with several parts of the specification. Therefore SEMST manages many-to-many relationships. A description of the links is shown in figure 5, where Si indicates a part of specification, Pi indicates a procedure/function in a program, and Ti indicates a set of test cases.

6.1.4.2 Links Establishment

In SEMST, the links are established while the test cases are entered into the system. Each test case in the system also contains two pointer elements which are used to point to the

[1]The term '*part of the specification*' is used to represent any subset of the specification which can be tested, such as a rule or a functionality described in the specification.

part of the specification and the part of the program linked with the test case. The links are represented by the item identifiers. If a test case has a link with a rule identified as *rule1*, the users need to type in rule1 when creating the link.

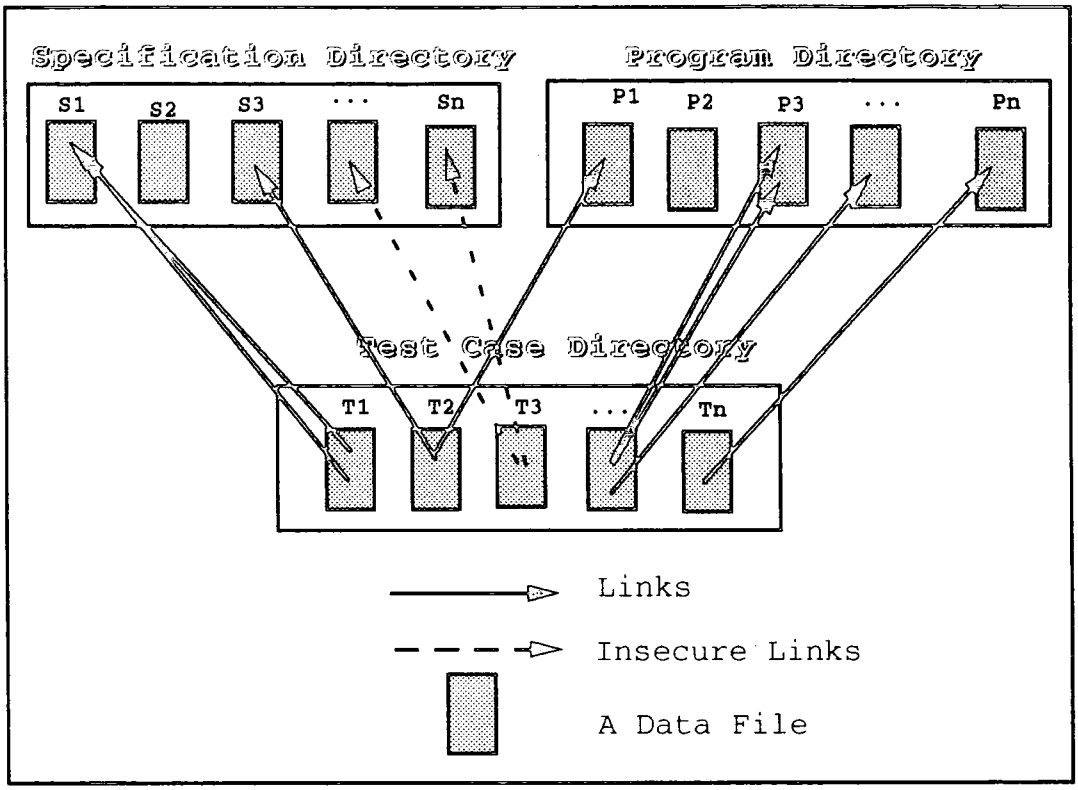

Figure 5. Internal Structure of the SEMST Database

6.1.4.3 Enquiry

SEMST allows the users to make an enquiry about links among the data in the system. When such an enquiry is made, the system provides the corresponding link information on the screen. With this mechanism, the users are able to know the current state of the links created between the data in the system. To operate this function, the users are asked to give the identifiers of the items whose link states are required.

6.1.4.4 Insecure Links

When an update has been conducted to a part of the specification or program, the links of the test cases with that part of the specification and program have become insecure. In order to keep the links among the data in the system up to date, SEMST provides a mechanism to control and manage these insecure links. The users can be made aware of which parts of the specification or program have been changed; which links have become insecure; and which test cases have been affected by the change. The insecure links can be changed to secure by modifying them.

6.1.5 Controlling Security

An attempt to update the same file by two persons at the same time is a dangerous activity, and will cause unexpected results. For example, suppose two people retrieve revision 2.4 of a file at the same time and modify it. Person A deposits his revision first, and person B somewhat later. Unfortunately, person B knows nothing about A's changes, so the effect is that A's changes are "undone" by B's deposit. A's changes are not lost since all revisions are saved, but they are confined to a single revision. SEMST prevents this conflict by locking. When a user wants to retrieve a revision from the system, the system will check it out and lock it so that the second user cannot retrieve the same file before a new revision of the file has been saved into the system. This function is also based on RCS.

6.2 System Architecture

6.2.1 Overview Of the System

SEMST is built on top of UNIX and RCS on a Sun workstation and has the abilities described in the previous sections. On the other hand, SEMST is designed as an integrated tool including interfaces with a *test case generator, static program analyser*, and *regression testing tool* etc.. Figure 6. shows an overview of the SEMST environment.

6.2.2 Functional Structure

SEMST provides its functionalities through four major components: *system monitor, specification management segment, program management segment* and *test case management segment*, which operate on the SEMST database. The functional structure of SEMST is shown in figure 7.

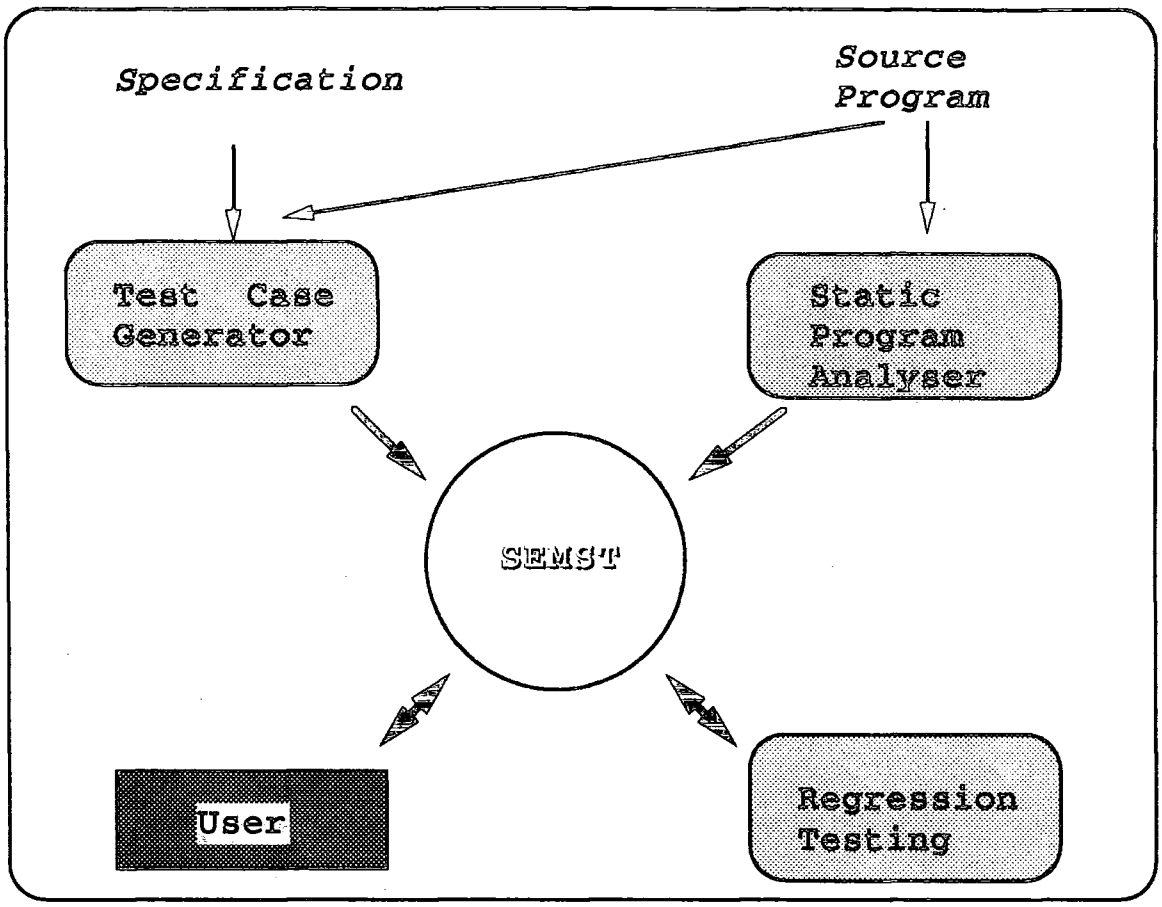

Figure 6. SEMST Environment Overview

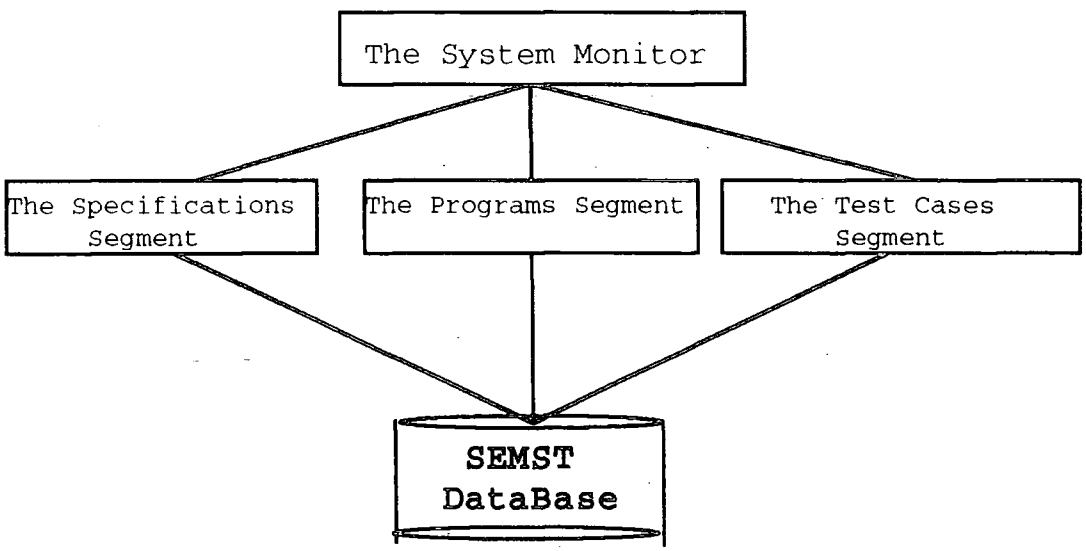

Figure 7. SEMST System Functional Architecture

The system monitor is responsible for controlling the whole system. Its functions include receiving and analysing the user commands selected from the system menu, and invoking the subsystem corresponding to the user command.

The specification management segment is used to control the user's access to program specifications. It involves several functions, such as aiding the user inputing a specification; retrieving or updating all versions of the specifications; and providing link state information.

The program management segment has the responsibility for controlling and managing the source code to be tested. The tasks include editing, storing, retrieving and updating all versions of a program file. It has been designed to include an interface with a static analyser and a mechanism to load the program static attributes tables into the system database. The purpose of these functions is to provide the data information to support the management of test cases used for program-based testing.

The test case management segment takes charge of controlling and managing all the activities with test cases. It provides the mechanisms to assist the user to input and update the test cases, to store the test cases into the system database, and to retrieve all versions of the test cases from the system database. It manages the links of test cases with specifications and programs.

From the above descriptions, five general functional areas associated with each segment can be categorised as follows:

- *input process.* This is concerned with actual inputs of the system (i.e. the input of specifications, test cases and programs).
- *output process.* This involves the output from the system. The system outputs are mainly a history of a test case/specification/program, link information and change information.

- *data maintenance.* This is relevant to the storage, update and retrieval of all versions of the data.
- *linkage management.* This involves establishing and controlling the links of test cases with specifications and programs.
- *security control.* This involves the control over the security of the data stored in the system database.

6.2.3 System Database

Whenever a new project enters into SEMST, the system creates three different directories, in each of which relevant type of the data associated with the project will be stored. The system database is built on the Unix file system.

6.2.3.1 Data Model

SEMST can be considered as an object-oriented database which supports an Entity-Relation-Attributes data model. It provides *structural object-orientation* concepts for the direct representation of real world entities with their complex structure. The objects in the database are the data files (specifications, test cases and programs) which are treated as entities with attributes and relationships. At the lowest level, this model is implemented as a collection of directories with each data class mapped onto a Unix directory. An object, which is an instance of a data class, is then implemented as a file within the directory.

When on-line, objects are held as C structures. However, to store them in the database, each object is output into a separate file. The database has two primitive interface functions for this purpose; one for reading objects from and one for writing objects to the database. To do this the functions must access the description of the appropriate C structure definition from a library of object descriptions. The data model is therefore easily

extended because all that is needed for the addition of an object to the data model is simply the creation of a new description in the object description library.

6.2.3.2 Data Representation

The format of specifications used in SEMST is shown in figure 8, in which the *Rule/Functionality Identifier* is an identifier used to identify each rule/functionality described in a specification file, and the *Rule/Functionality Description* is a context describing the contents of rule/functionality corresponding to the identifier.

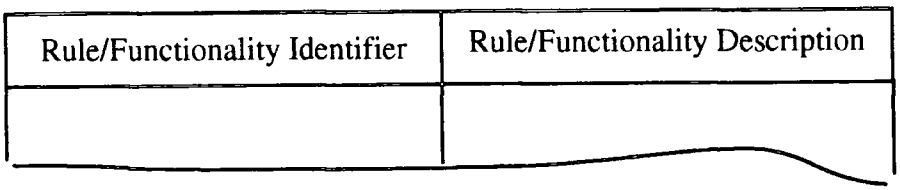

Figure 8. Logical Structure of a rule/functionality description in SEMST

There is a literal definition of the test case [81] that states a test case must consist of two components: a description of the input data and a description of the output for that set of input. The representation of test cases in SEMST is an adaptation of this definition which is shown in figure 9, where, the *Identifier* is used to identify a test case, the *Inputs* is the description of input data and the *Outputs* is the description of expected output, the *Strategy-base* is used to represent the testing strategies(known as Specification-based or Program-based) on which the test case is constructed, the *Link with P* is a pointer used to point to the part of the programs linked with the test case and the *Link with S* is a

pointer used to point to the part of the specifications to which the test case is relevant.

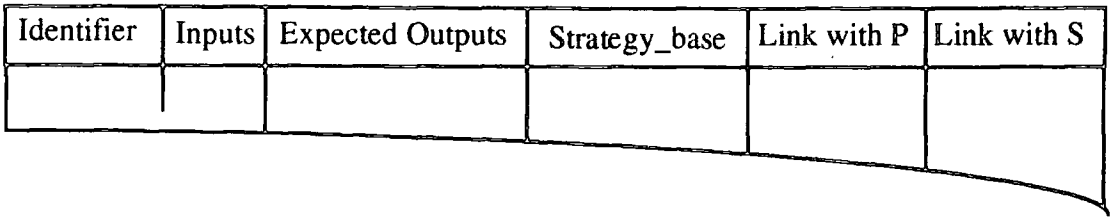

Figure 9. Logical Structure of a test case in SEMST

The program attribute tables in SEMST are the same format as those tables in a static program analyser. This analyser should be able to produce the module, path, branch and statement tables associated with a program. Generally, a module table consists of an identifier of a procedure or function (normally the name of procedure or function is used) and a statement number indicating the place of the module in the system. A branch table should contain a branch number used as an identifier of the branch and a description of the branch (normally the numbers of statement sequence are used). It is the same idea for the format of the path table and statement table.

The object descriptions stored in the library are shown below:

```
typedef struct spec_file
                        {
                          char rule_ident[15];
                          char rule_content[15];
                          struct spec_file * next;
                        } SF, *PSF;

typedef struct spec_management
                        {
                          char filename[16];
```

```
                    char rule_ident[15];
                    char secure_mark[3];
                    struct spec_management * next;
                  } SMT, *PSMT;

typedef struct Test_Case_file
                  {
                    char TC_ident[15];
                    char input_data[15];
                    char expected_output[15];
                    char TC_strategy[3];
                    char P_link[20];
                    char S_link[20];
                    char TC_type[3];
                    struct Test_Case_file *next;
                  } TCF, *PTCF;

typedef struct Test_Case_management
                  {
                    char filename[16];
                    char TC_ident[15];
                    char secure_mark[3];
                    struct Test_Case_management *next;
                  } TCMT, *PTCMT;

typedef struct program_name_management
                   {
                    char filename[16];
                    char proc_func_name[15];
```

```
                    char secure_mark[3];
                    struct program_name_management *next;
                   } PNMT, *PTPMNT;

    typedef struct testcase
                   {
                    char ident[15];
                    struct testcase *next;
                   } TC,*PTC

    typedef struct links
                  {
                   char link[15];
                   struct links *next;
} LINK, *PLINK;

PSF psf_head, psf_tail;
PSMT psmt_head, psmt_tail,pro_psmt;
PTCF ptcf_head, ptcf_tail;
PTCMT ptcmt_head, ptcmt_tail;
PTPMNT pmnt_head, pmnt_tail;
PTC ptc_head;
```

6.3 User Interface

SEMST is an interactive system. A menu is provided to the users to operate the system. There are two kinds of menus in SEMST. The *system menu (or main menu)*, indicated in figure 10, is used to guide the users into the subsystem; and the *subsystem menu* is used

to list the functions of each subsystem from which the users can select an appropriate function. SEMST contains three subsystems, namely the specification segment, the test case segment and the program segment, each of which includes the same function menu shown in figure 11. To use SEMST, the users do not need to learn any new command languages.

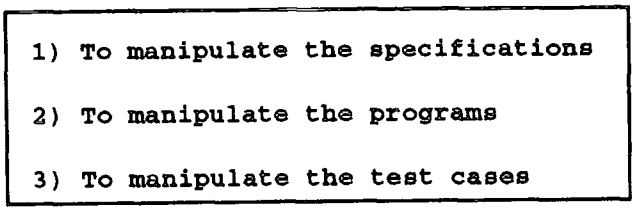

Figure 10. The System Menu

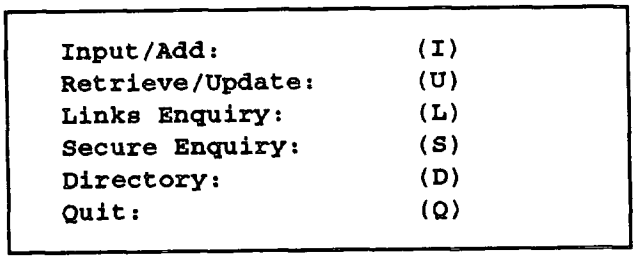

Figure 11. The Subsystem Menu

6.4 An Example

SEMST has been implemented as a prototype system to examine its utility in an industrial environment, it has been used to manage the testing documentation associated with part of a Tunnel-Control system developed by Marconi Command and Control System in 1985. The System Functional Specification in the Tunnel-Control project is referred to as an example of a rule-based specification which is composed of a set of rule descriptions, and the Test Case Specification document in the Tunnel-Control project is referred to as an example of the test cases relevant to the rule-based specification. This section illustrates several outputs from the SEMST. Appendix A describes the use of SEMST in more detail.

Suppose that the specification document, consisting of a set of rule descriptions identifiered as *rule0, rule1, rule2, ..., rulen* and the test case document, consisting of a set of descriptions of the test cases identified as *test00, test01, test02, ..., testmn* have been loaded and stored in the SEMST database.

The *Retrieve/Update* function in the specification subsystem of SEMST provides the user with the ability to retrieve or update a rule/functionality specified in the specification. Figure 12 shows a rule description retrieved from the system database. The description of the rule is identified as "rule5", and its file name is "d_rule5".

The *Link Enquiry* function in the specification subsystem of SEMST helps the users to obtain the information about the linkage of specification with test cases. Figure 13 illustrates what test cases are linked with "rule1", "rule2", "rule3", "rule4", "rule8", and "rule9".

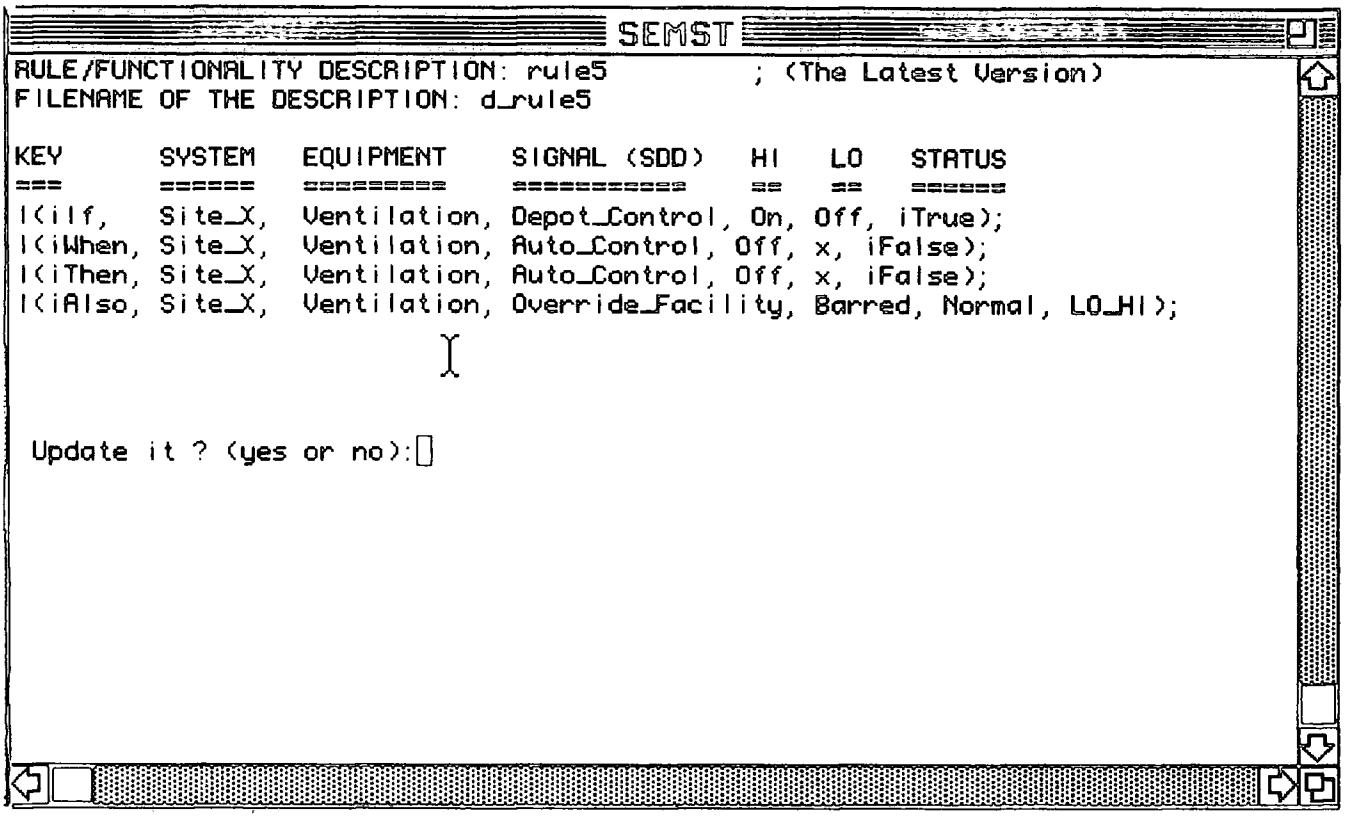

Figure12. The Display Of a Retrieved Rule Description

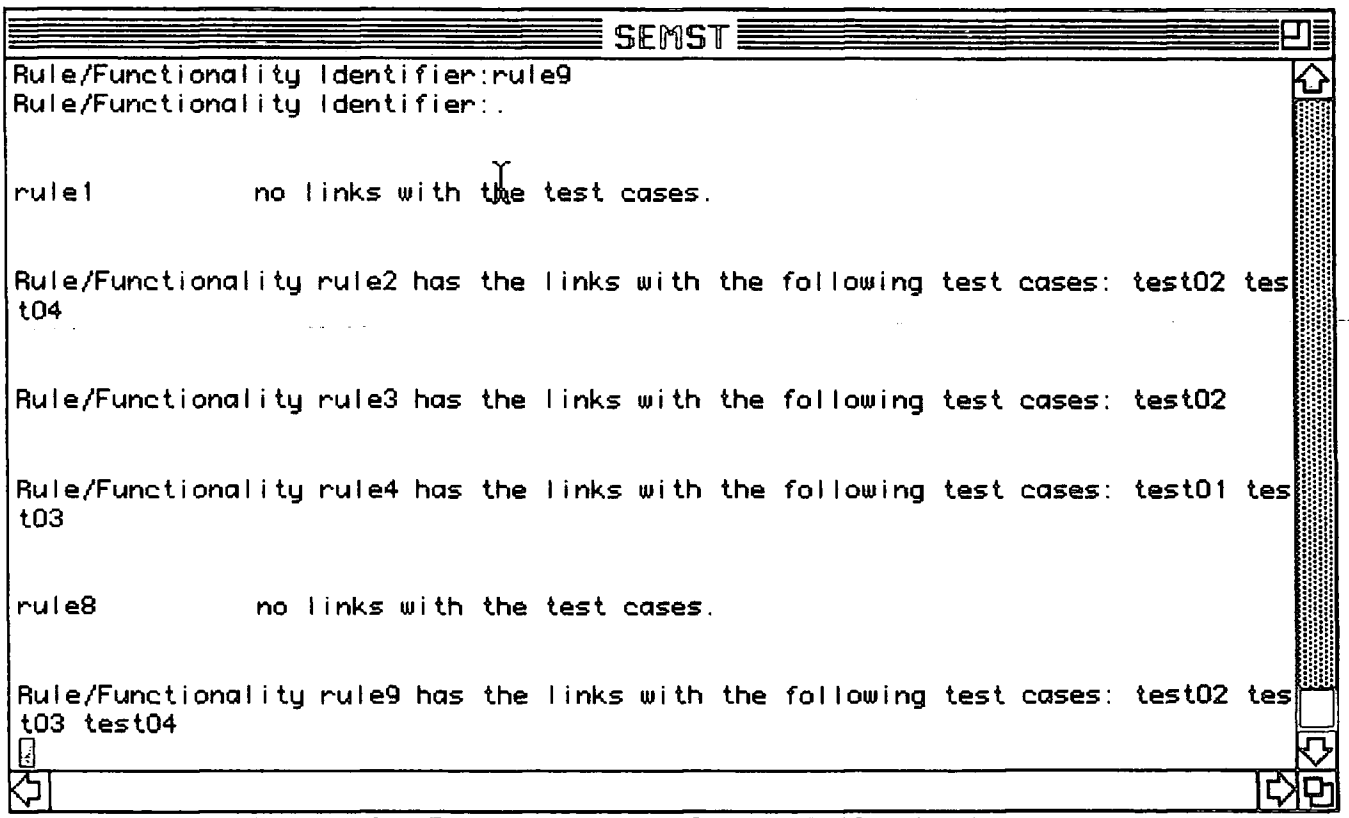

Figure13. Links Between the Rules and the test cases

The *Secure Enquiry* function in the specification subsystem of SEMST provides the user with the information about what rule/functionality described in the specification has been changed and what links have become insecure due to the change. Figure 14 indicates that the "rule3", "rule5", and "rule7" are modified so that the links of these rules with test cases may be insecure. Insecure links can become secure after proper changes to them.

Similar to the functions provided in the specification subsystem of SEMST, the test case subsystem also provides the functions to retrieve or update a test case record in the system database, and to make the enquiries about the links and change information.

Figure 15 shows a retrieved test case record whose identifier is "test04". The first five lines in the figure shows the attributes of test case record "test04". After that it is a display of the contents of the input data file and expected output file.

Figure 16 shows the links of "test01", "test02" and "test04" with the specification and the program.

Figure 17 indicates the insecure links of the test cases with the rules in the specification.

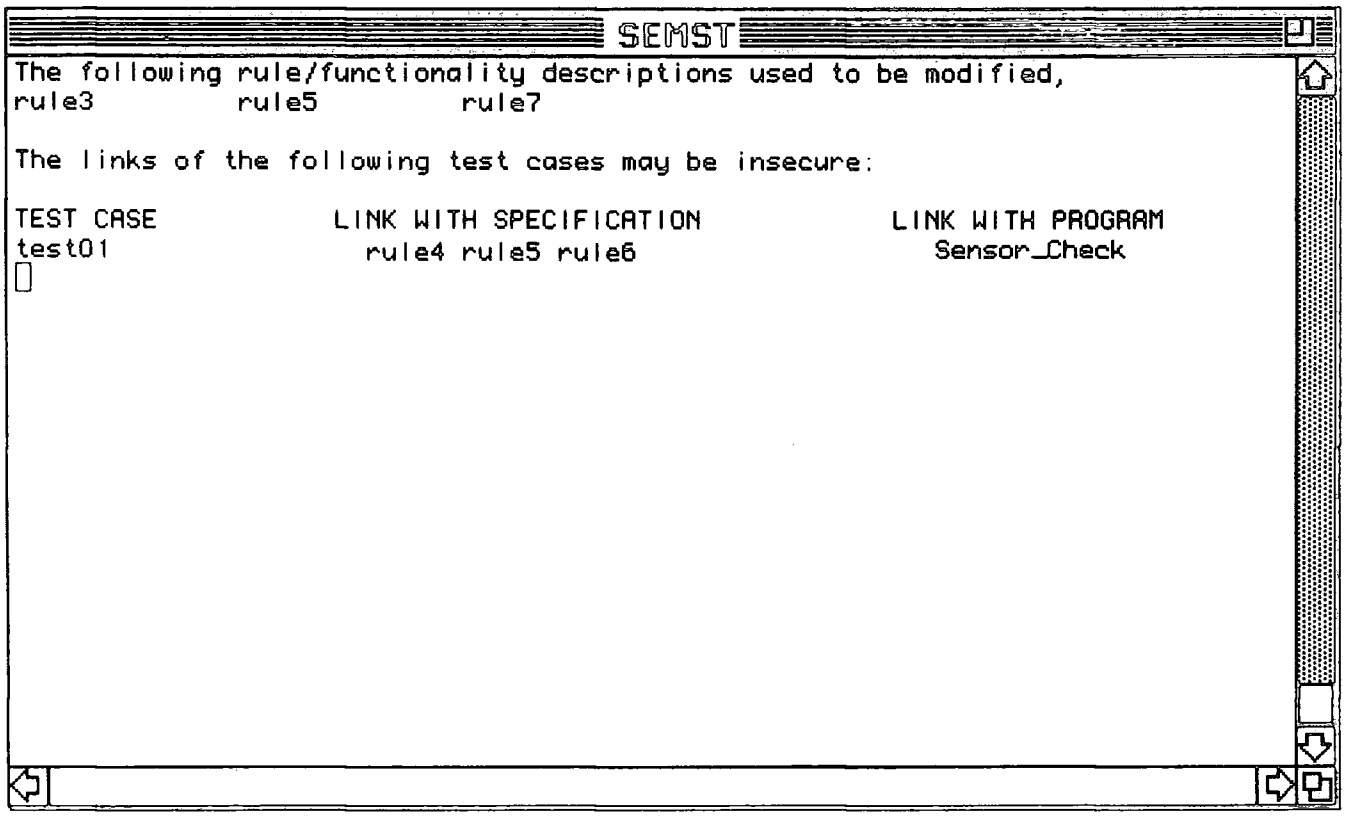

Figure 14 . An Example of the Secure links Checking in the Specification Segment

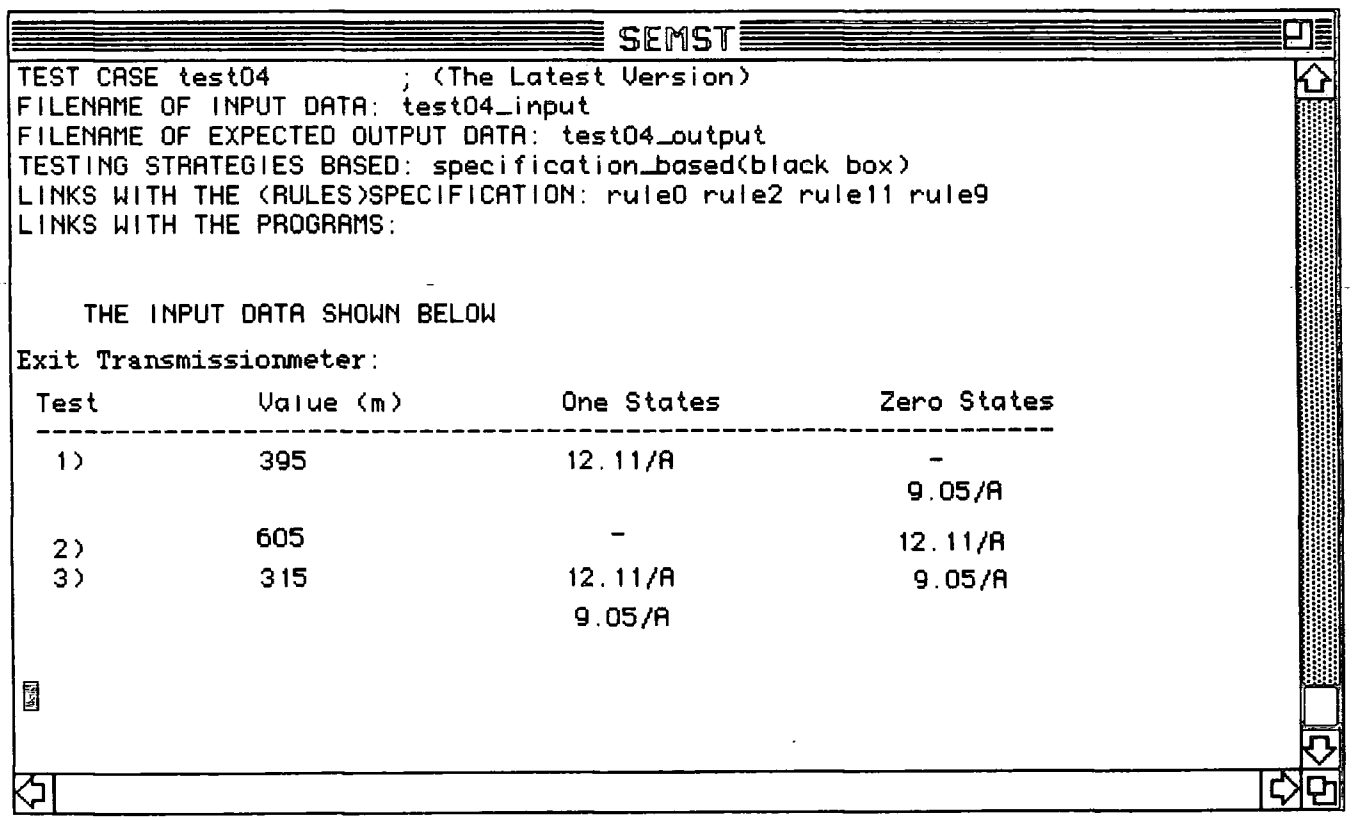

Figure 15 . The Display of a Retrieved Test Case Record

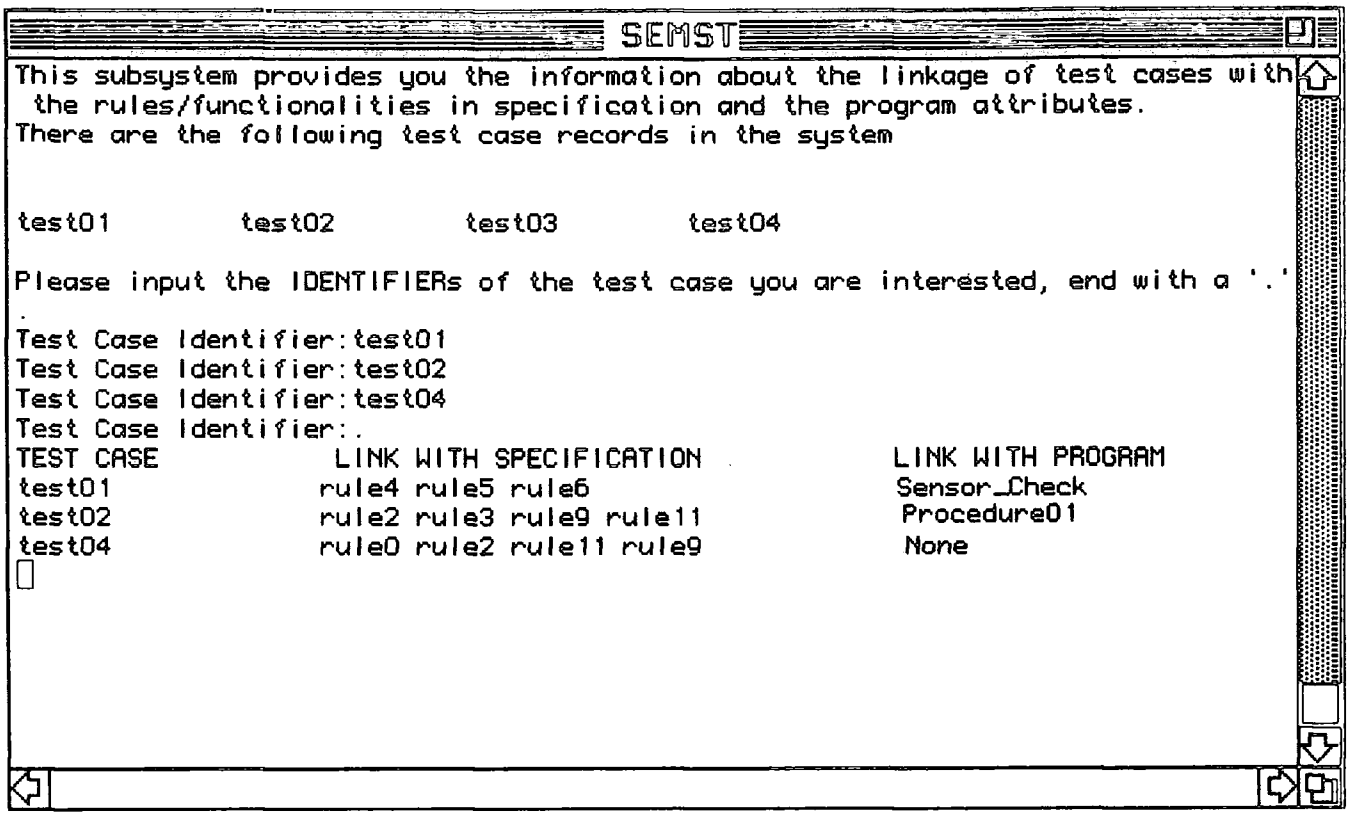

Figure 16 . Links Between The Test Cases and the Rules

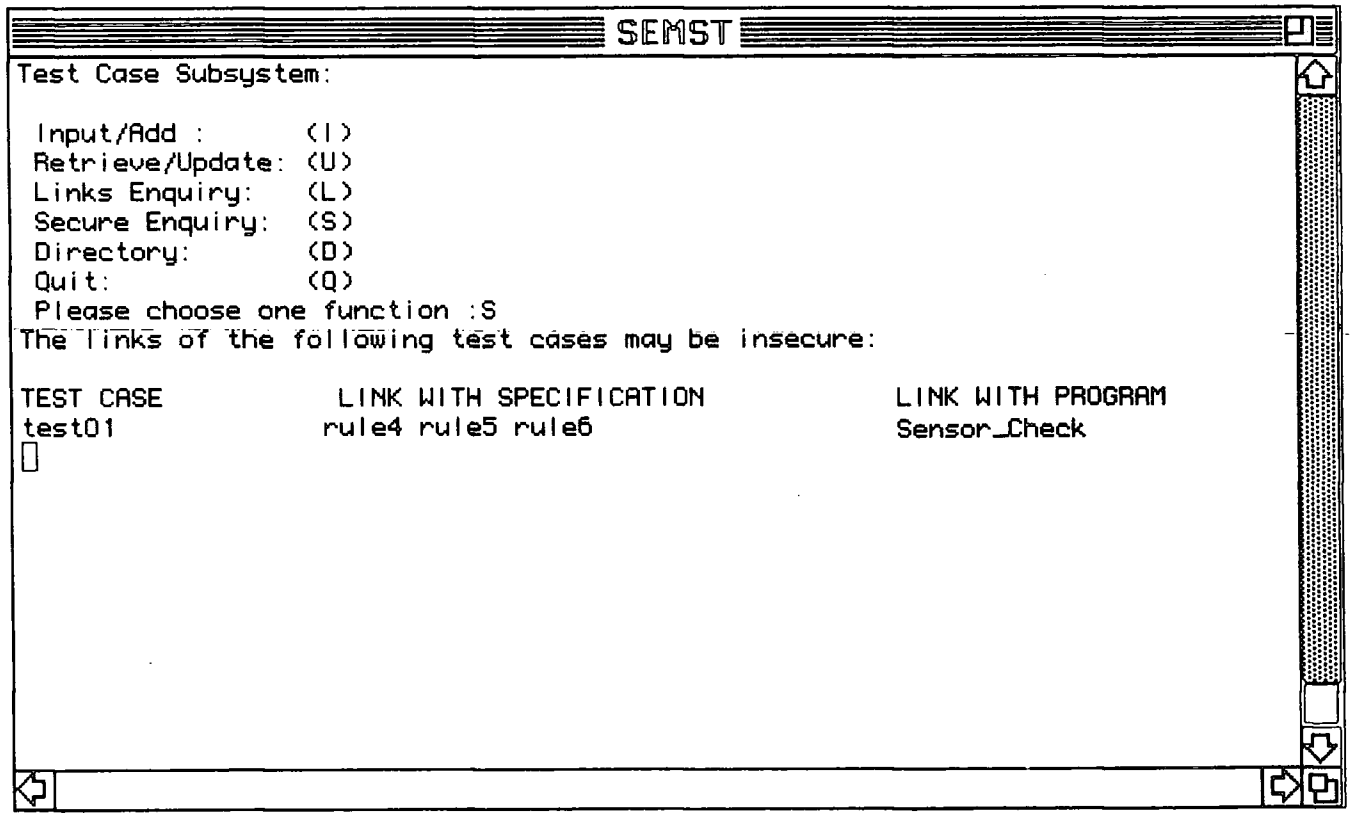

Figure 17 . An Example of the Secure Links Checking in The Test Case Segment

6.5 The Design of SEMST

SEMST is designed as a database environment which manages and maintains test data in the project life cycle. Section 6.2, the section on the system architecture, has been devoted to describing how the prototype SEMST has been designed to achieve its functional requirements. This section is used to clarify and complement several points about the design of the SEMST system.

6.5.1 The System Functional Structure

From functional structure point of view, the SEMST system consists of five major parts which have been shown in figure 7:

1. the system monitor;
2. the specifications segment;
3. the programs segment;
4. the test cases segment; and
5. the SEMST database.

The *system monitor* acts as a main control program in the SEMST system, whose major functions are:

- to analyse the user commands and call the relevant subsystems;
- to create a new database(i.e. new directory) for a new project user; and
- to save the system information into the database before exiting SEMST system.

The above three *segments* are the *subsystems* of SEMST, and each of them is functionally independent of the other. The *SEMST database* is designed as a central repository where the test data and the analysed information are stored, and shared by these subsystems. It is the common data area on which the independent subsystems are able to communicate with each other.

6.5.2 The SEMST Database

The Database Model

As described in section 6.2.3, the SEMST can be thought of as an object-oriented database management system. The specifications, test cases and programs used in the real world are represented using the Entity-Relation-Attribute data model and stored as the files with data records in the SEMST database. The objects in the SEMST database are files, namely specification files, test case files and program files[see section 6.2.3.1].

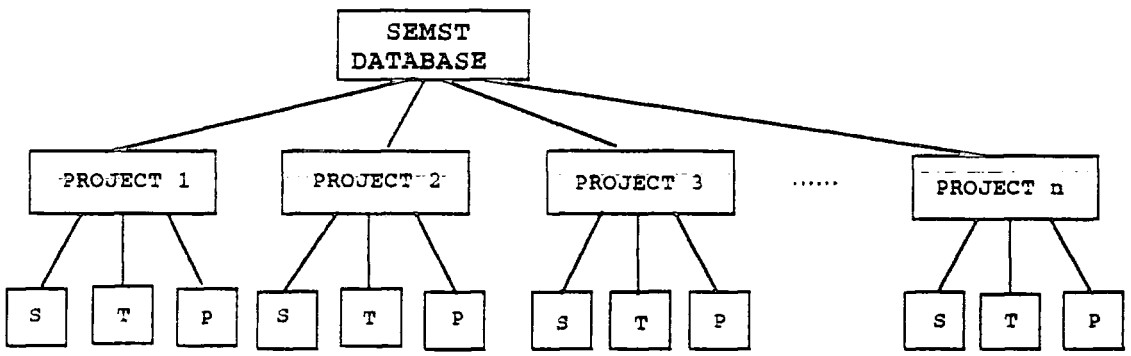

Figure 18. The Structure of SEMST Database

The Database Structure

Since SEMST is implemented in the UNIX environment, its structure is tailored to the Unix file system which is a hierarchical architecture. As shown in Figure 18, the SEMST database directory can be separated into a number of subdirectories in terms of the project users. Each project directory comprises three data areas: *specification area*(i.e. the "S" in figure 18), *test case area*(i.e. the "T" in figure 18); and *program area*(i.e. the "P" in figure 18). In these three areas, the data are maintained under RCS control(e.g. all versions of the test cases are stored in area "T"). In addition, each of these areas comprises a *management table*, whose purpose is to register each of the new data items loaded into the database (e.g. in area "T", a test-cases management table(TCMT) is stored to record every name of the new *test case record* or *test case file* [2]). The management tables mentioned above are main data structures in SEMST, based on which the system is manipulated.

6.5.3 The Links in SEMST

SEMST manages the links between test data based on the identifiers associated with each test data item. SEMST requires that each test data item be given a name as its identifier. For instance, a rule/functionality specified in the specification must have a name, and this name will be contained in the test case records if these test cases are based on such rule/functionality. The links are established when the test cases are entered into the SEMST database. The links can be changed when the test cases are modified.

Guide [11] is a hypertext tool available in the Unix on the Sun workstations. However, link management is poor in *Guide*, and it is unsuitable for satisfying the SEMST functional requirements. It is unrealistic to develop a hypertext system for managing the links within the time constraints of this project. For these reasons, the prototype version of SEMST

[2] A test case file stores a set of test case records, and each test case record is identified by a test case name. See Appendix A for more details.

uses the identifiers concept as a case study of managing the links.

6.6 The SEMST Properties

The capabilities of the prototype version of SEMST have been described in section 6.1. This section firstly summarises the results that SEMST has achieved, and then gives a further description of applying SEMST to a real project – the Tunnel Control System, which has been presented in section 6.4.

6.6.1 Highlights of the SEMST Achievements

1. **Loading Data.** SEMST consists of a mechanism for helping the user input the test data into its database. The user can be guided by the SEMST system instructions to input the data. The vi screen editor is provided for inputing and editing the new data files. If the data files are already in the machine, SEMST can convert these files into its database.

2. **Maintaining Versions and Releases.** SEMST keeps track of the changes to the files and maintains all versions of the files in its database. Given this version history, four vital questions can be answered:

   - what changes were made,
   - who made the changes,
   - when were the changes made, and
   - why were the changes made.

   SEMST allows the user to define the releases in order to baseline a number of versions of test data.

3. **Retrieving and Updating.** The user is allowed to retrieve any version of the data files from the SEMST database and to update the data information in the SEMST system. In SEMST, the links between the test data can also be updated. The reasons for providing this functionality are the following:

   - The previous links established between the data may be wrong.
   - When a new function is added to the specification or program, there may be a new link that should be created between the added function and the previous test cases.
   - When a modification has been made to a part of the specifications or programs, the current links of test cases with the modified specification or program have become insecure. A change of the insecure links may have the effect of changing these links back to secure.

4. **Managing Links.** The links of test cases with specifications and programs are controlled and maintained in association with each version of these components. SEMST provides the user with information about the insecure links resulting from the changes to the specification or program. From this information, the user can become aware of the affected test cases because of specification or program changes. After a modification is made to the insecure links, SEMST can change the links to secure status automatically.

5. **Controlling Data Security.** SEMST prevents multiple users from updating the same file simultaneously. When a user has retrieved a file from the system, this file is then locked by SEMST to stop the second user retrieving it again until the first user's task is finished.

6.6.2 Application of SEMST To the Real Project System

The prototype version of SEMST has been used to manage the test documents in the Tunnel-Control system[3]. A demonstration of such application has been presented in section 6.4. This section is intended to give a more detailed explanation about the use of SEMST with the Tunnel-Control System.

Relevant Features of the Tunnel-Control system

The following illustrates some relevant features of the Tunnel-Control system, which should be considered when applying SEMST to it:

- the System Specifications in the Tunnel-Control system are rule-based, and written in a Pascal-like language,
- the test cases specified in the Test Case Specifications in the Tunnel-Control system are functional test cases.
- the outputs from the execution of the system may be the inputs to the next execution of the system. Therefore, the test cases have been selected from part of the system's outputs.

The SEMST Approach

SEMST is able to handle the above features associated with the Tunnel-Control system, by doing the following:

- defining an identifier for each rule described in the System Specification,

[3]The Tunnel-Control System is a real-time system for controlling and managing the various conditions in a road Tunnel. The system was developed by Marconi Command and Control System.

- defining an identifier for each test case specified in the Test Case Specification,
- defining an identifier for each test case selected from part of the system's outputs,
- defining a file for each set of the relevant test cases(e.g. all the test cases used for testing ventilation function should be put in one file),
- defining a file for the test cases selected from the part of system execution outputs,
- if a test case is used to test a rule specified in the System Specification, there is then a link between this test case and rule,
- if the execution of a test case traverses a part of the program(e.g. some branches in a program module), there is then a link between such test case and part of program.

Figure 12 has shown a retrieved rule specification from the SEMST database, whose identifier is "rule5" and filename is "d_rule5".

Figure 15 has shown a retrieved test case from the SEMST database. The first six lines in the figure are the header of this test case, which list: the name of this test case(i.e. "test04"); the name of its input data file (i.e. "test04_input"); the name of its expected_output file(i.e. "test04_output"); the testing strategy it is based on(i.e. "specification_based"); its links with specification(i.e. the rules("rule0", "rule2", "rule11" and "rule9") linking with it) and its links with programs(this test case has no link with the programs). The test case record displayed in figure 15 is incomplete. The user can see the complete test case by typing "return".

Figure 14 has shown that due to the change of "rule3", "rule5" and "rule7" the links of test case "test01" with the specification("rule4", "rule5" and "rule6") and the program(the Sensor_Check subroutine) have become insecure. Hence, the test case "test01" is the affected by the specification changes.

Other figures have been explained in section 6.4.

6.7 Testing SEMST

The development process of the SEMST system has passed phases of analysis, specification, design, implementation and testing. Each of these phases has produced the appropriate documents. This section describes how the SEMST system has been tested during its development.

6.7.1 Review development Documents

In the phases of requirement analysis, functional specification and design of the SEMST system development, a review was used as the testing method to examine the development deliverables/documents. The development documents were reviewed to be correct with respect to the system requirements.

6.7.2 Unit/Module Testing

During the implementation of SEMST, each module in the system was tested first. The test cases chosen for testing the modules were based on both functional/specification-based and structural/program-based testing strategies. Therefore each module was tested functionally and structurally. The objective of unit testing was to ensure that each module in the SEMST system satisfied its design requirements.

6.7.3 Integration/Subsystem Testing

After the module testing, the qualified modules were integrated into the subsystems of SEMST. When testing the subsystems, the test cases were selected mainly for checking

the interfaces between modules and examining the functionalities of each subsystem. The subsystems were tested against the subsystem design.

6.7.4 System Testing

During system testing, interfaces between the subsystems of SEMST were checked, and the communication between the subsystems on the basis of the SEMST database were examined. The test documents associated with the Tunnel-Control system were used as the test cases for system testing SESMT. The results of the system testing indicated that the prototype SEMST satisfied its functional requirements.

Chapter 7

Assessment and Conclusion

Introduction

This chapter concludes the dissertation by presenting a review of the SEMST system. Three sections are included in the chapter. Section 7.1 presents an assessment of the prototype version of SEMST and also discusses the future research and development directions. Section 7.2 gives an overview of the major topics addressed in the dissertation. A summary of this dissertation is included in the last section.

7.1 Assessment of SEMST and Future Work

SEMST is one kind of data management system used for supporting the testing process. It has the following important features which distinguish it from similar testing management tools.

- Storage, retrieval and update of the specifications, programs, and test cases.
- Maintenance of the versions of these components
- Baseline of a number of these component versions.
- Management of the relationships among these components.
- Traceability after a modification is made to one of the items.

At present, SEMST is a prototype version consisting of 2,500 lines of C code. As a prototype, it inevitably has some limitations such as the length of a file name is limited to 15 letters and a non-text file cannot be updated in SEMST. Possible extensions to SEMST have been considered and are described below:

- Firstly, the support for complex interrelationships among the data should be extended in SEMST. At the moment SEMST can only support the management of the relationships of test cases with specifications and programs.
- Secondly, the present SEMST prototype uses its own file formats and is not actually able to share or interchange data with other life cycle tools, although it is required and designed to be able to do so. Therefore, in the future an interface model should be explored to furnish an integrated environment.
- Thirdly, RCS provides the configuration management support by defining a configuration as a set of revisions and checking the revisions out according to a certain

criterion. The criteria include the *default* (the latest version), the *release-based* (a release or a branch number), the *author-based and state* (author name or state attribute), the *date-based* (the date), and the *name-based* (the symbolic name). The SEMST prototype adopts the first two as the criteria used for retrieving a version of a file. Later development of SEMST should add the rest of the aspects in improving the configuration management.

- Fourthly, to improve the configuration management support, SEMST should also provide a facility to prevent the released items in the system database from being casually modified.

- Fifthly, the programs and their attributes in SEMST are designed to be well formatted, and language-independent. However, this can be achieved only when a generic static program analyser is provided. Since there is neither a generic static program analyser nor a specific language static analyser implemented on the Sun workstation, the features of storage of program's attributes and the management of the linkage of test case with program attributes remains undeveloped in the prototype version of SEMST.

- Finally, future work is also needed to provide a user-friendly interface in SEMST such as windows and graphics.

7.1.1 Further Review of the SEMST System

This section is used to present a further analysis and evaluation of the SEMST system based on a comparison of SEMST with some other existing database systems.

7.1.1.1 Advantages of SEMST

The key concept in the SEMST system is its attempt to apply SCM techniques to the testing process and to provide traceability between test data(e.g. the specifications, test cases and programs) across the system life cycle. One important, distinguishing feature of SEMST is its ability to support the retesting process which is performed in association with the evolution of software system. In summary, the SEMST system would be beneficial to the testing process in the following:

- A way of computer-aided test data management is provided;
- All versions of the data items can be maintained, so that the details of change to each test data item(e.g. when and what the changes are made) can be recorded;
- The baseline for a set of test data items can be defined and managed;
- The links between test data can be controlled(e.g. creating, enquiring and modifying the links);
- By managing the links, the following can be identified:
  - derived data items to their sources;
  - activated parts by the execution of data items; and
  - change effect on the data items and their links (e.g. the affected parts by the change, and the links possibly no longer valid between the changed data items(i.e the insecure links)).
- The modification of links has the advantage of keeping the state of data relationships up to date.
- By locking a retrieved file from the system, the shared data and simultaneous update problems[section 2.1.3] may be solved.

Many existing database systems claim their cooperation with a hypertext system, so that the relationships between the life-cycle data can be managed. In contrast to SEMST, however, the current hypertext systems do not involve the identification of change effect on the data and the relationships between the data. There is no mechanism provided in a hypertext system to manage the insecure links which could possibly result from the data modifications. Moreover, the current hypertext systems have not addressed the question of ensuring the reliability of the links established in their systems. Hence, few of these database systems are sufficient for supporting the evolution of software and the revalidation process.

7.1.1.2 Limitations of SEMST

The limitations of the current version of SEMST have been identified as follows:

- The support for configuration management in SEMST is only based on the facilities provided in RCS, and therefore SEMST has the limitations which exist in RCS. For example, RCS does not provide a mechanism to prevent a released(baselined) data item from being randomly modified. But according to SCM discipline, any baselined data items should not be modified unless it has been agreed by the board of SCM.
- It is not actually able to communicate with other life-cycle tools.
- It does not provide control for all possible links between the specifications, the test cases and the programs;
- In comparison with the hypertext systems, SEMST does not support for non-linear data relationships control(which is provided in the hypertext systems). On the other hand, the hypertext systems are designed to manage a wider range of data, including the storage of graphic, audio and video data applications. However, SEMST has only considered the support for text data. RCS has the same problem.

- The current SEMST uses the concept of an identifier to manage the links between the data items. However, this approach may be less effective than the approach provided in the hypertext systems. In the hypertext systems, a graph concept is used to deal with the relationships between data components. The graph is a collection of the nodes and the links, and each node and link(i.e. the edge between the nodes) in the graph are given a name. The users are allowed to traverse the nodes and links on the graph. A graphic representation of data and relationships is usually much easier to understand than other forms of data output.
- The current version of SEMST has not provided a systematic approach to ensure that the links(original or changed) created between the data are accurate to reflect the data relationships, although it provides certain assistant mechanisms to allow the users to check the created links and to update the links.
- The current SEMST system does not name the links, so that the kind of relationships existing between the linked data are not clear.
- As a prototype, it is ineffective for a large amount of test data. For instance, the output information on the computer screen will become messy when there are a lot of the information. There is no printed data output provided in the SEMST prototype.
- The present SEMST system is weak in query operations compared to other database management systems(e.g relational database).
- SEMST needs to improve its user interface.

7.1.2 Lessons Learnt From the Research

The following presents a summary of what has been learnt from doing this research project.

- **The View of Software Testing.** Software testing has evolved from being viewed as a follow on activity after the coding to being viewed as a continuous activity

performed in each phase of the software life cycle. Recently, the formal transformation process has been accepted as a software development process model[104], which is intended to support the generation of correct programs from the specifications. However, this model still stays at a theoretical level and is unlikely to be used in practice in the near future. Therefore, at the present stage, testing is still widely believed to be a pragmatic verification and validation mechanism for ensuring a high quality software production. The testing process has been represented using a test life cycle model which is embedded within the whole software life cycle[section 3.2].

- **The Application of SCM to Testing.** SCM, as a software engineering discipline, is concerned with the management and control of the evolution of software system. The testing process is associated with the evolution of software system. This could be understood from the following. Firstly, when a software error is found by a testing activity, a modification of the specification or program or both will occur. On the other hand, after the change to the specifications or programs, a relevant testing activity will require to take place in order to ensure the changed system is correct with respect to its requirements. Furthermore, the test data components have close relationships between each other; a change of one component could affect the validity of other relevant components. Finally, the revalidation of changes is a major task in software maintenance. It requires the reselection of test cases and the update of the previous test plan. Because of the above features, the testing process should be subject to the control of SCM. Of the SCM methods, change control, version control and record-keeping/traceability are relevant to the management of test data [section 4.4].

- **The Development of SEMST.** The application of SCM to testing deserves individual emphasis, but it has been overlooked in past years. In this research, the SEMST system has been designed and developed as a practical model of combining the testing process with SCM methods. The application of SEMST would augment current testing or maintenance support environments in a way that supports the testing activities associated with software changes. Little such help has been given

in previously developed database systems. Hence, the development of SEMST is significant, although the present version of SEMST has some weakness.

- The Way of Doing Research. Undertaking a research project is also a vehicle for learning the way of doing research. Generally, a research project involves the activities of investigation, analysis, design, implementation and evaluation. From doing this research, especially by rewriting the original version of the dissertation, the lesson has been particularly learnt in how to sensibly conduct analysis and evaluation of a research deliverable. It has been realised that without an appropriate analysis, the research will not lead to progress.

7.2 Overview of the Major Topics of the Dissertation

This dissertation has presented a description of research undertaken on the topic of software testing management.

The background and purpose of this research have been introduced in the first chapter.

The research approach involves the investigation of software configuration management methods and its application to the testing process; the study of software testing techniques and methods; the exploration of the significance of software testing management; the survey of related work in the past; and the development of a new environment – SEMST.

In the dissertation, software testing has been viewed as the continuous activity and task of planning, designing and constructing tests, and of using those tests to assess and evaluate the quality of work performed at each stage of the software development life cycle.

The dissertation has concluded that software configuration management is the complete mechanism for controlling and recording the status of all deliverables, their relationships and their changes. In practice, it is difficult to construct a cost-effective and usable

software system without a good system of configuration management in place. All software items concerned with software testing should be subject to configuration management control.

The dissertation has discussed software testing methods and techniques based on the classification of testing into specification-based and program-based testing. It has concluded that these two testing strategies are basically complementary approaches to software testing; one of them can not be used to replace the other.

The dissertation has given a description of testing in the software life cycle and an emphasis on the early test planning. It has also addressed the needs for the management of software testing by discussing three major problems associated with software testing:

- difficulties in early planning for testing;
- large amount of data produced during the testing process; and
- testing software changes.

The dissertation has pointed out that the support of software configuration management will help to solve the above problems.

The examples of other existing systems described in this dissertation indicate that many environments provide certain management assistance but few of them provide enough information to enable the management of the testing process.

SEMST, a support environment for the management of software testing, has been described in the dissertation as a practical model resulting from the research.

A number of shortcomings of the current version of SEMST have been discussed which are considered as the future development directions.

7.3 Summary of the Dissertation

This dissertation has been devoted to addressing a significant issue namely to apply software configuration management methods to the testing process. The SEMST system presented in this dissertation is an implementation of managing the testing process with SCM support. It is believed that the application of SEMST will enhance the scope of software testing environments with respect to control and management of testing changes. However, the current SEMST is a prototype and many improvements are needed.

In summary, the design of integrated testing support environments that covers a full spectrum of management activities in an integrated manner will need further attention before they become reality for practical use. The support environments need to take into account not only the management of test data, like the features of the current SEMST, but also other management activities(e.g. test resources management, test plan and schedule management). The support environments should also provide facilities for reflecting both the evolutionary nature of the software development(i.e. change to software component, software system structure and process, plans and schedules, etc.) and the relevant testing activities associated with such evolutionary features.

The dissertation is now concluded with the wish that one day more advanced solutions, techniques and tools would be explored to solve the problem area that this dissertation has considered.

Appendix A

How to use SEMST

SEMST helps the users to maintain all the versions of *specifications, test cases and programs* associated with a particular project, as well as to manage the relationships among these components. The characteristics of SEMST are described in the main body of this dissertation. This section focus on explaining the procedures to use SEMST. In this section, the examples used to illustrate a rule-based specification and the test case descriptions are from the *Tunnel Control* project documents developed by Marconi Limited in 1985.

A.1 Enter the System

The format of the command for entering SEMST is as follows:

% semst *ProjectName*

Suppose a user wants to use SEMST to manage the project entitled *Tunnel-Control*, then the user should type "semst Tunnel-Control". For a new project which has not previously been entered into SEMST, the system will create three new directories for storing the data associated with this new project. To do this, the system will ask the users if they are sure they want to create a new project in SEMST and the user should answer "yes"(user can simply type 'y'), otherwise it will quit from SEMST.

A.2 Manipulating the Subsystems

To operate the functions of the subsystem the user selects a number relevant to the subsystem in the main menu. For instance, if the users want to operate the functions in the test case subsystem, the users should type "3" after the SEMST prompt.

Each subsystem has the functions listed in the menu shown in figure 11, where:

- the **Input/Add** is the function for loading the data (e.g. descriptions of specifications and test cases) into the SEMST database, including the input of a new file and addition of new data items to an existing file. When a new project is entered into the SEMST, this function should be selected first;
- the **Retrieve/Update** is the functions that perform the retrieval of any version of data items from the database and control the update to this item when required by the users;

- the **Links Enquiry** is the function used to help the users to get the information about the current state of the links among the data items in the system;
- the **Secure Enquiry** function helps the users to know what part of specification has been changed and what links have become insecure;
- the **Directory** is the function listing all names associated with the files stored in the directory. All files are under the control of RCS;
- the **Quit** is the function provided for exiting the subsystem.

A.3 Specification Manipulation

The users operate the functions in the specification segment by selecting "1" from the system menu. The functions listed in this subsystem menu are summarised in the above section. In this section more detailed descriptions are given on how to operate each of them.

A.3.1 Input/Add

In order to input a specification, the user should choose "I" from the menu provided in the specification segment. The system provides the users with a mechanism to input their specifications with respect to the project by a set of rule/functionality descriptions with the relevant rule/functionality identifiers and to store these descriptions in a number of files. This kind of file is called the *Specification File* in SEMST. There is another kind of file in SEMST called *Rule/Functionality File* which contains the descriptions of a particular rule/functionality. This kind of file may be constructed outside of SEMST. SEMST can convert these files into its database when the users give the complete path name of the file. On the other hand, the users are allowed to input a new rule/functionality file with

the vi screen editor when the users give a name of the rule/functionality file. The format of a specification file is shown in figure 8.

The procedure of inputing/adding the specification descriptions is guided by the system, and the options are illustrated bellow:

1. To input a name of the specification file
2. To input a rule/functionality identifier.
3. To input a name of the rule/functionality file. If the file exists, go to step 5, otherwise go to the next step.
4. To input the contents of rule/functionality description.
5. If more rule/functionality descriptions are required to input, go to step 2, otherwise end the procedure.

A.3.2 Retrieve/Update

By selecting "U" from the menu in the specification segment, the user can conduct the operations of retrieval and update. There are two ways offered by SEMST to retrieve a version of a rule/functionality description from the specification files stored in SEMST database. One way is to retrieve by the *Filename* of the specification file. Another way is to retrieve by the *rule/functionality identifiers*. When the first method is chosen, the system will display all the names of specification files in the system relevant to the user project. The users should then select the name of a file to be retreived and input it into the system. After that, the system will show all the rule/functionality identifiers included in this file on the screen. Then the user can retrieve a rule/functionality description from the database by inputing a rule/functionality identifier and a version number into the system. Similarly, all of the rule/functionality identifiers in SEMST database will be displayed

on the screen after the user chooses the second method to retrieve the rule/functionality descriptions, and the retrival is then same as described above.

The contents of a rule/functionality description can be shown on the screen after it has been retrieved from the system database, see figure 12 for an example. In this figure, the display of a rule description whose file name is "d_rule", identifiered as "rule5", is reflected. If the users want to make a modification to the description contents, they type "yes" after SEMST prompt. Then the system will provide the vi screen editor to assistant the modification.

A.3.3 Links Enquiry

This function in the specification subsystem helps the users to obtain the information about the linkage of the specification with test cases. After the users choose "L" from the menu, the system will firstly display all of the rule/functionality identifiers in SEMST relevant to the user's project, and then ask the users to input the interested identifiers one by one. When the users finish the input, the system will show the relevant information about the links on the screen. Figure 13 shows what test cases are linked with "rule1", "rule2", "rule3", "rule4", "rule8" and "rule9".

A.3.4 Secure Enquiry

This function provides the users with the information about what rule/functionality described in the specification files has been modified and what links have become insecure due to the modification. Figure 14 shows an example.

A.3.5 Directory

This displays all of the names of the files stored in the specification directory of SEMST including the specification files and rule/functionality files. The files are all under RCS control.

A.4 Test Case Manipulation

This section describes how to operate the functions provided in the test case subsystem of SEMST.

To begin this function, select "3" from the main menu. In the test case directory of the SEMST database, there are also two kinds of files, namely the *Test Case File* and the *Input data* or *Expected Output File*. A test case file is used to store a set of *test case records* consisting of the following attributes:

- the test case identifier,
- the input data,
- the expected outputs,
- the testing strategy based,
- the testing coverage criterion used, and
- the links with the specification and the program.

The format of a test case record is shown in figure 9. An input data or expected output file contains a description of the inputs or expected outputs associated with a test case record. This kind of file may be constructed outside of SEMST. SEMST is able to load

these files into its database when the user gives the complete path name of the file. If the file dose not exist, the system will provide a vi screen editor to help the users to input a new input data or expected file as long as the users give a file name to the system.

A.4.1 Input/Add

The users wanting to operate the function to input/add the test case into SEMST should select "I" from the menu provided in the test case subsystem. The whole procedure of inputing/adding the test cases into SEMST is to input each test case record into various test case files, while the input/ add of a test case record includes the input of each attribute associated with the test case mentioned above.

In SEMST, the attribute named "Strategy-Base" is defined as a necessary item for a test case record and is used to represent the testing strategies on which the test case is based. It is widely recognized that there are two classes of the testing strategies used in software testing: *Specification-Based* and *Program-Based.* In SEMST, the users should use "S" to represent the specification-based strategy use "P" to represent the program-based strategy and use "SP" for the both of strategies. The users are required to input the *test coverage criterion* when "P" is selected for the strategy- based. In this case, "s" should be used to represent satement coverage, "f" should be used to represent the module testing (or unit testing), "b" should be used to represent the branch coverage and "p" should be used to represent the path coverage.

At the stage of inputing/adding, the links between the test cases and the specification or the program can be established. To do this, the users should give the relevant identifiers of the rule/functionalities or the program attributes. On the whole, the procedure to input/add the test cases into the SEMST database can be summarized as follows:

1. To input a name of the test case file.

2. To input a test case identifier.

3. To input the attributes of a test case record.

4. If more test case records are required to be input, go to step 2, otherwise end of the procedure.

A.4.2 Retrieve/Update

From the menu in the test case segment, "U" should be selected to start this function. Similar to the procedure of retrieving/updating a rule/functionality description described in section A.3.2, the users can retrieve a version of a test case record by checking the test case files or test case identifiers and then give the version's number to be retrieved. When a version of the test case record is retrieved from the system database, the contents of its each attribute will be displayed on the screen. Figure 15 shows a retrieved test case record whose identifier is "test04". The first five lines in the figure shows the attributes of test case record "test04". After that it is a display of the contens of the input data file and expected output file.

The users are allowed to modify every attribute of a test case record after it has been retrieved from the system database. The modification of a test case record is divided as four parts: the modification of input data, the modification of expected outputs, the modification of testing strategy and the modification of the links with the specification and program. All these functions can be guided by the system to operate, and vi is provided as a screen editor to help the modification.

When a modification has been undertaken to the links between a test case and the specification or program, the system will make secure mark to these link states. As a result, if the modified link states were previously insecure, it will become secure due to the modification.

A.4.3 Links Enquiry

This function is provided to help the users get the information about the linkage of the test cases with the specification and program. After the users choose "L" from the function menu, the system will display all of the test case identifiers in SEMST associated with the users' project, and then will ask the users to input the identifiers one by one. When the users finish the input, the relevant link information will be shown on the screen. Figure 16 shows the links of "test01", "test02" and "test04" with the specification and the program.

A.4.4 Secure Enquiry

By selecting "S" from the function menu in the test case segment, the users can obtain what links have become insecure due to the modification taken on the parts of the specification. Refer to figure 17 for an example.

A.4.5 Directory

This function lists all of the names of the files stored in the test case directory of SEMST including the test case files and the input data/ expected outputs files. These files are controlled by the RCS.

Bibliography

[1] Akscyn, R. M., McCracken, D. L. and Yoder, E.A., *A Distributed Hypermedia System for Managing Knowledge in Organisations*, Communications of the ACM, 31 (7), pp. 820-835, 1988.

[2] Babich, Wayne A., *Software Configuration Management Coordination for Team Productivity*, Addison-Wesley, Reading M.A. 1988.

[3] Bazelmans, Rudy, *Evolution of Configuration Management*, ACM Sigsoft Software Engineering Notes, Vol.10 No.5, pp. 37-46, Oct 1985.

[4] Beizer, Boris, *Software Testing Techniques*, -2nd ED., New York: Van Nostrand Reinhold, 1990.

[5] Bernstein, Philip A., *Database System Support for Software Engineering – An Extebded Adstract*, 9th International Conference on Software Engineering, pp. 166-178, Monyerey, USA, March 30 - April 2, 1987.

[6] Bersoff, Edwrad H., Handerson, Vilas and Siegel Standley G., *Software Configuration Management*, Prentice Hall, Englewood Cliffs, N.J., 1980.

[7] Bicevskis, J., Borzovs, J., Stranuiums, U., Zarins, A., and Miller, E.F., *SMOTL – A System to Construct Samples for Data Processing Programming Debugging*, IEEE Transactions on Software Engineering, SE-5, (1), pp.60-66, 1979.

[8] Bigelow, J., *Hypertext and CASE*, IEEE Software, Vol. 5, No. 2, pp. 23-7, March 1988.

[9] Bott, M.F., *ECLIPSE - An Integrated Project Support Environment*, Peter Peregrinus, London, UK, 1989.

[10] Boyer, R.S., Elpas, B., and Levit, K.N., *SELECT - A Formal System for Testing and Debugging Programs By Symbolic Execution*, Proceedings of International Conference on Reliable Software, pp. 234-244, April 1975.

[11] Brown, Peter J., *Turning Ideas Into Products: The Guide System*, Proceedings of the Hypertext '87 Workshop, The University Of North Carolina, pp.30-40,, Novermber 1987.

[12] Budd, T.A. and Demillo, R.A., Lipton, R. J., and Sayward, F. G., *Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Program*, 7th ACM Symposium on Principles of Programming Languages, January 1980.

[13] Bull, GEC, ICL, Nixdorf, Olivertti, Siements, *PCTE: A basis for a Portable Common Tool Environment*, Functional Specification, Fourth Edition, 1986.

[14] Clarke, L.A., *A System to Generate Test Data and Symbolically Execute Programs*, IEEE Transactions on Software Engineering, SE-2, (3), pp.215-222, 1976.

[15] Clarke, C.A. and Hassel, J., *A Close Look at Domain Testing*, IEEE Trasactions on Software Engineering, Vol. 8, No. 4, July 1982.

[16] Collofello, James S. and Woodfield, Scott N., *A Proposed Software Maintenance Environment*, IEEE 1984.

[17] Conradi, Reidar and Normann, gerhard, *CM for Distributed and Heterogeneous Systems*, ACM SIGSOFT, Software Engineering Notes, Vol.13, no.4, pp.69-70, Oct. 1988.

[18] Clarke, Lori A., Richardson, Debra J., and Zeil, Steven J., *TEAM: A Support Environment for Testing, Evaluation, and Analysis*, ACM Sigsoft/Plan Software Engineering Symposium on Practical Software Development Environment, November 1988.

[19] Coward, P David, *A Review of Software Testing*, Information and Software Technology, Vol. 30, No.3, April 1988.

[20] Daly, E.D., Management of Software Development, IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

[21] Dam, A. van, *Hypertext '87 keynote address*, Comm. ACM 31 (7), pp. 887-895, 1988.

[22] Demillo, Richard A., McCracken, W. Michael, Martin, R.J. and Passafiume, John, *Software Testing and Evaluation* , Software Engineering Research Centre, Georgia Insititute of Technology, Library of Congress Cataloging-in-Publication Data, August 1986.

[23] DeRemer, F. and Kran, H.H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. SE-2, pp.80-86, June 1976.

[24] Department of Defense, Military Standard, MIL-STD-480, *Configuration Control - Engineering Changes, Deviations, and Waivers*, 1968.

[25] Department of Defense, Military Standard, MIL-STD-483, *Configuration Management Practices for Systems, Equipment, Munitions and Copmputer Programs*, 1970.

[26] Donohoo, J. D. and Searingen, D., *A Review of Software Maintenance Technology*, Rome Air Development Centre, RADC-TR-80-13, Interim Report, Feb. 1980.

[27] Dowson, M., *ISTAR - An Integrated Project Support Environment*, Sigplan Notices: Proceedings of 2nd SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, pp. 27-33, Jan. 1987.

[28] Duran, J. W. and Ntafos, S., *An Evaluation of Random Testing*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 438-444, July 1984.

[29] Elmendrof, W.R., *Functional Analysis Using Cause-Effect Graphs*, Proceedings of SHARE XLIII, SHARE, pp. 567-577, New York, 1974.

[30] Estublier, Jacky, *Configuration Management: the Notion and the Tools*, International Workshop on Software Version and Configuration Control, Grassau, pp. 38-61, January 1988.

[31] Estdale, J., Ostrolenk, G., Atles, A. and Younger, E., *The System Description Database*, The REDO handbook, A Compendium of Reverse Engineering for Software Maintenance, Edited by Zuylen, Henk van, 2487-TN-WL-1027, Version 0.8, pp. 261-285, August 1991.

[32] Evans, Michael W., *Productive Software Test Management*, John Wiley & Sons, Inc., 1984.

[33] Failrley, Richard, *Software Engineering Concepts*, McGraw-Hill International Editions, 1985.

[34] Feldman, S.I., *Make – A Program for Maintaining Computer Programs*, Unix Programmer's Supplementary Documents, Vol.1, 4.3 Berkely Software Distribution, Virtual VAX-11 Version, University of California, Berkeley, California, April, 1986.

[35] Fernstrom, Christer and Ohlsson Lennart, *ESF – An Approach to Industrial Software Production*, Software Engineering Environments:Research and Practice, Edited by Keith Bennett. Published by Ellis Horwood Limited, pp. 17-28, 1989.

[36] Fischer, K.F., Raji, F. and Chrusciki, A., *A Methodlogy for Re-Testing Modified Software*, National Telecomms Conference Proceedings, pp. B6.3.1-6, Nov. 1981.

[37] Floyd, R. W., *Assigning Meaning to Programs*, Proceedings of the Symposia in Applied Mathematics, Vol. 19, pp. 19-32, 1967.

[38] Foster, John, *Software Maintenance*, Technical Report, British Telecommunications Plc., 1989.

[39] Gamalel-Din, Shehab A. and Osterweil, Leon J., *New Perspectives on Software Maintenance Process*, IEEE Conference Software Maintenance, Phoenix, Arizona, pp. 14-22, October 1988.

[40] Garg, Pankj K. and Scacchi, Walt, *On Designing Intelligent Hypertext Systems for Information Management in Software Engineering*, Hypertext '87 papers, pp. 409-432, November 1987.

[41] Gehani, N., and McGettrick, A.D., *Software Specification Techniques*, Addison-Wesley Publishing Company, 1986.

[42] Goldberg, A., *The Influence of an Object-Oriented Language on the Programming Environment*, Interactive Programming Environments, New York, pp. 141-174, 1984.

[43] Gourlay, John S., *A Mathematical Framework for the Investigation of Testing*, IEEE Transactions on Software Engineering, SE-9(6):666-709, November 1983.

[44] Goodenough, J.B. and Gerhart, S.L., *Toward a Theory of Test Data Selection*, IEEE Transactions on Software Engineering, SE-1(2):156-173, June 1975.

[45] Habermann, A. Nico and Notkin, Dave, *Gandalf: Software Development Environments*, IEEE Transactions On Software Engineering, Vol. SE-12, No. 12, December 1986.

[46] Halasz, F. G., *Reflections on NoteCards: Seven Issues For the Next Generation of Hypermedia System*, Communications of the ACM, 31 (7), pp. 836-852, 1988.

[47] Hartmann, J. and Robson, D.J., *Regression Testing with Hypertext Support*, Technical Report - 1989, University of Durham, 1989.

[48] Hartmann, J. and Robson, D.J., *Techniques for Selective Revalidation*, IEEE Software, pp. 31-36, January 1990.

[49] Hetzel, William, *Program Test Methods*, Englewood Cliffs, N.J. Prentice-Hall, 1973.

[50] Hetzel, William, *The Complete Guide to Software Testing*, Second Edition, QED Information Sciences, Inc., 1988.

[51] Howden, William E., *A Functional Approach to Program Testing and Analysis*, IEEE Transactions on Software Engineering, Vol. SE-12. No. 10, October, 1986.

[52] Howden, W.E., *Symbolic Testing and the DISSECT Symbolic Evaluation System*, IEEE Transactions On Software Engineering, SE-3, July 1977.

[53] Howden, W.E., *An Evaluation of the Effectiveness of Symbolic Testing*, Software Practice and Experience, Vol. 8, pp. 381-397, July 1978.

[54] IEEE, *IEEE Standard for Software Configuration Management Plans*, Std. 828-1983, 1983.

[55] IEEE Std. 729-1983, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Press, 1983.

[56] IEEE Standards Board, *Software Engineering Standards*, ANSI/IEEE, Std. 1008-1987.

[57] Jensen, Randall W. and Tonies, Charles C., *Software Engineering*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1979.

[58] Jiang, J. Liu, L. and Robson, D. J., *Research Exisiting Tools and Techniques for Dynamic Analysis, Generation of Test Case, Linkage of Test Cases With Specifications and Programs, and Selection of Test Cases to Rerun*, Esprit 2487 REDO report, 2487-TN-DU-1008, 30 March 1990.

[59] Jones, C.B., *Systematic Program Development Using VDM*, Secon Edition, Prentice Hall International(UK) Ltd, 1990.

[60] Jonson, D.W., *The Development Facility Approach to Improved Software Development*, AFIPS Conference Proceedings of the National Computer Conference, pp. 235-239, May 1981.

[61] Kaiser, Gail E. and Habermann, A. Nico, *An Environment for System Version Control*, IEEE Computer, Vol. Feb., pp. 415-420, 1983.

[62] Kaiser, G.E., Feiler, P.H., and Popovich, S.S., *Intelligent Assistance for Software Development and Maintenance*, IEEE Software, pp.40-49, May 1988.

[63] Kemmerer, Richard A., *Testing Formal Specifications to Detect Design Errors*, IEEE Transctions on Software Engineering, SE-11(1), January 1985.

[64] Kenning, Rachel J. and Munro, Malcolm, *Configuration Management - The State of the Art*, Technical Report, Centre for Software Maintenance, University of Durham, 1989.

[65] King, J.C., *Symbolic Execution and Program Testing*, Communications of the ACM, Vol. 19(7), pp. 385-394, July 1976.

[66] Lampson, B.W. and Schmidt, E.E., *Organizing Software in a Distributed Environment*, In Proceeding of the ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, pp. 1-13, June 1983.

[67] Laski, Janusz W. and Korel, Bogdan, *A Data Flow Oriented Program Testing Strategy*, IEEE Transactions on Software Engineering, SE-9(3): 347-354, May 1983.

[68] Laski, Janusz W., *Testing in Top-Down Program Development*, Second Work Shop on Software Testing and Verification, and Analysis, pp.72-79, July 1988.

[69] Leblang, David B. and McLean, Gordon D. Jr., *Configuration Management For Large-Scale Software Development Efforts*, Workshop on Software Engineering Environments For Programming in the Large, pp. 122-127, June 1985.

[70] Leblang, David B, Chase, R.P.Jr., and Spike, H., *Increasing Productivity with a Parallel Configuration Manager*, International Workshop on Software Version and Configuration Control, Grassau, pp.21-37, January 1988.

[71] Leung, Hareton K.N. and White, Lee J., *A Study of Regression Testing*, Technical Report TR 88-15, Department of Computer Science, The University of Alberta, Edmonton, Alberta, Canada, September 1988.

[72] Lewis R., Beck D. W., Hartmann J. and Robson D. J., *Assay - A Tool To Support Regression Testing*, BTRL/ Dept. of Computer Science, Durham, Technical Report, 1988.

[73] Liu, Lung-Chun and Horowitz, Ellis, *Object Database Support For A Software Project Management Environment*, Proceedings of the Third ACM SIGSOFT/SIGLAN Software Engineering Symposium on Practical Development Environments, pp. 85-96, 1988.

[74] Loo, P. S. and Tsai, W. K., *Random Testing Revisited*, Information and Software Technology, Vol. 30, No. 7, pp.407-417, September 1988.

[75] Lutz Mike, *Testing Tools*, IEEE Software, pp.53-57, May 1990.

[76] Marchionini G. and Shneiderman, B., *Finding Facts vs. Browsing Knowledge in Hypertext Systems*, IEEE Computer, Vol.4, No.12,pp.70-80, Jan. 1988.

[77] Military Standard, *Common Ada Programming Support Environment (APSE) Interface Set (CAIS), DOD-STD-1836* , DOD-STD-1836, 9 October 1986.

[78] Miller, Edward *Introduction to Software Testing Technology*, Tutorial: Software Testing & Validation Techniques, Second Edition, Edited by Miller, E. and Howden, W.E., The IEEE Computer Society Press, pp. 1-16, 1981.

[79] Mohammed, A.H., *Hypertex in Testing Support*, M.Sc Dissertation, Department of Computer Science, University of Durham, 1988.

[80] Morgan, Carroll, *Programming from Specifications*, Prentice Hall International Series in Computer Science, 1990.

[81] Myers, G.J., *The Art of Software Testing*, July 1979.

[82] Maur, P. and Randell, B., *Software Engineering: A Report On a Conference Sponsored By the NATO Science Committee*, NATO, 1969.

[83] Ntafos, Simeon C., *On Required Element Testing*, IEEE Transactions on Software Engineering, SE-10(6):795-803, November 1984.

[84] Neighbors, J., *The Draco Approach to Constructing Software From Reusable Components*, IEEE Transactions on Software Engineering, V. SE-10, No. 5, September 1984.

[85] Ostrand, Thomas J. and Balcer, Marc J., *The Category-Partition Method For Specifying and Generating Functional Tests*, Communication of the ACM, Vol.31, No.6, pp. 676-686, June 1988.

[86] Ould, Martyn A., *Testing In Software Development*, Cambridge University Press, 1986.

[87] Panzl, D.J., *Automatic Software Test Drivers*, Computing, pp. 44-50, April 1978.

[88] Parikh, G., *Handbook of Software Maintenance*, John Wiley and Sons publishers, 1986.

[89] Perry, D. and Kaiser, G., *Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems*, Proceedings of the ACM Fifteenth Annual Computer Science Conference, 1987.

[90] Pirie, I., *Benefits of Automating Configuration Management*, Notes 1st Software Maintenance Workshop, Durham, England, 1987.

[91] Powell, Patricia B., *Software Validation, Verification, and Testing Technique and Tool Reference Guide*, NBS Special Publication 500-93, U.S. Government Printing Office, 1982.

[92] Pressman, Roger S., *Software Engineering - A Practitioner's Approach*, Second Edition, McGraw-Hill International Editions, 1987.

[93] Ramamoorthy, C.V. and Ho, S.F., *Testing Large Software With Automated Software Evaluation System*, IEEE Transactions On Software Engineering, SE-1, No.1, pp.157-169, March 1975.

[94] Rapps, Sandra and Weyuker, Eline J., *Selecting Software Test Data Using Data Flow Information*, IEEE Transactions on Software Engineering, SE-11(4):376-375, April 1985.

[95] Reedy, A., Stephenson, Dudar, E. and Blumberg, F.C., *Software Configuration Management Issues in the Maintenance of Ada Software Systems*, IEEE Computer, pp. 234-245, 1989.

[96] Richardson, D.J. and Clarke, L.A., *Testing Techniques Based on Symbolic Evaluation*, Software Requirements, Specification and Testing, Proceedings of CSR Workshop, University of East Anglia,10-12 April 1984, Edited by T. Anderson.

[97] Richardson, D.J. and Clarke, L.A., *Partition Analysis: A Method Combining Testing and Verifications*, IEEE Transactions on Software Engineering, SE-11(12): 1477-1490, December 1985.

[98] Richardson, D.J., *Approaches to Specification-Based Testing*, Software Engineering Notes, Volume 14, No. 8, Proceedings of the ACM Sigsoft'89 Third Symposium on Software Testing, Analysis and Verification(TAC3), Edited by Richard A. Kemmerer, December 1989.

[99] Richardson, D.J. and Thompson, M.C., *Test Data Selection Using The RELAY Model of Error Detection*, Proceedings of 5th Annual Pacific NW Conference on Software Quality, 1987.

[100] Rochkind, Marc J., *The Source Code Control System*, IEEE Transactions on Software Engineering, Vol. SE-1, No.4, pp. 364-370, December 1975.

[101] Schneidewind, N.F., *The Use Of Simulation in the Evaluation of Software*, Computer, pp. 47-53, April 1977.

[102] Shooman, Martin L., *Software Engineering*, McGraw-Hill International Book Company, 1983.

[103] Sommerville, Ian, *Software Engineering*, Second Edition, Addison-Wesley Publishing Company, 1985.

[104] Sommerville, Ian, *Software Engineering*, Third Edition, Addison-Wesley Publishing Company, 1989.

[105] Softool Corporation, *Configuration Control CCC/DM Turnkey Dependency and Build Option, Users Manual*, 1989.

[106] Spivey, J.M., *Understanding Z – A Specification Language and Its Formal Semantics*, Cambridge University Press, 1988.

[107] Stucki, L.G. and Foshee, G.L., *New Assertion Concepts for Self-Metric Software Validation*, Proceedings of IEEE Confernece on Reliable Software, Los Angeles, CA, pp. 59-65, April 1975.

[108] Tanaka, A., *Equivalence Testing For Fortran Mutation System Using Data Flow Analysis*, Georgia Institute of Technology, Department of Information and Computer Science, 1981.

[109] Taylor, Richard N., Belz, Frank C., Clarke, Lori A., Osterweil, Leon, Selby, Richard W., Wileden, Jack C., Wolf, Alexander L., and Young, Michal, *Foundations For The Arcadia Environment Architecture*, Proceedings of the Third ACM SIGSOFT/SIGLAN Software Engineering Symposium on Practical Development Environments, pp. 1-13, 1988.

[110] Teitelbaum T., *A Tour Through Cedar*, IEEE Transactions on Software Engineering, SE-11 (3): 285-302, 1985.

[111] Teitelman, W. and Masinter, L., *The Interlisp Programming Environment*, Computer, 14(4): 25-33, April 1981.

[112] Tichy, Walter F., *Design, Implementation, and Evaluation of a Revision Control System*, Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, pp. 58-67, 1982.

[113] Tichy, Walter F., *An Introduction to the Revision Control System*, Programmer's Supplementary Documents, Vol.1, 4.3 Berkely Software Distribution, Virtual VAX-11 Version, University of California, Berkeley, California, April, 1986.

[114] Tichy, Walter F., *Smart Recompilation*, ACM transactions on Programming Languages and Systems, Vol. 8, No. 3, pp.273-291, July 1986.

[115] Tichy, W.F., *Tools for Software Configuration Management*, International Workshop on Software Version and Configuration Control, Grassau, pp. 1-20, January 1988.

[116] Venkatramani, K. and Clark, R., *Using Configuraed Directories to Solve Software Maintenance Problems*, Proceedings of Conference on Software Maintenance, Phoenix Arizona, pp. 172-177, October 1988.

[117] Scacchi, Walt and Stadel, Manfred, *CM for Non-Textual Representation*, ACM SIGSOFT, Software Engineering Notes, Vol. 13, No.4, pp. 71-73, Oct. 1988.

[118] Weyuker, E.J. and Ostrand, T.J., *Theories Of Program Testing and the Application of Revealing Subdomains*, IEEE Transaction on Software Engineering, SE-6, 3, pp, 236-246, May 1980.

[119] White, L.J. and Cohen E.I., *A Domain Strategy for Computer Program Testing*, IEEE Transaction on Software Engineering, Vol. 3, pp. 247-257, 1980.

[120] White, L.J., *Software Testing and Verification*, Advances In Computers, Vol.26, pp. 335-391, Orlanao, Fla: Academic Press, 1987.

[121] Whitley, D.J., *The Benefits of Automated Configuration Managment*, Scicon Limited.

[122] Wileden, Jack C., Wolf, Alexander. L, Fisher, Charles D. and Tarr, Peri L., *PGRAPHITE: An Experiment in Persistent Typed Object Management*, Proceedings SIGSOFT '88: Third Symposium on Software Development Environments, Dec. 1988.

[123] Winkler, F.H. and Stoffel, C., *Version Control in Families of Large Programs*, Proceedings 9th International conference on Software Engineering, Monterey, California, March 1987.

[124] Winkler, F.H., *Report on the First International Workshop on Software Version and Configuration Control*, ACM SIGSOFT, Software Engineering Notes, Vol.13, No.4, pp.61-63, 1988.

[125] Yankelovich, N., Meyrowitz N. and Dam, A. van, *Reading and Writing the Electronic Book*, IEEE Computer 18 (10), pp. 15-30, 1985.

[126] Yau, S. S. and Kishimoto Z., *A Method for Revalidating Modified Programs in the Maintenance Phase*, IEEE COMPSAC 87 Int. Conf. Procs. pp.272-277, Tokyo, Japan, 1987.

[127] Zeil, S.J., *Perturbation Testing for Computation Errors*, Seventh International Conference on Software Engineering, March 1984.

[128] Zeil, S.J., *Selectivity of Data-Flow and Control-Flow Path Criteria*, Proceeding of the Second Workshop on Software Testing, Verifcationm and Analysis, pp. 216-222, Banff, Canada, 19-21 July 1988.